Lähdekoodiplagiarismin automaattisesta tunnistamisesta

Jussi Ampuja

Tampereen yliopisto
Informaatiotieteiden yksikkö
Tietojenkäsittelyoppi
Pro gradu -tutkielma
Ohjaajat: Pentti Hietala ja Erkki Mäkinen
Toukokuu 2015

Tampereen yliopisto
Informaatiotieteiden yksikkö
Tietojenkäsittelyoppi
Jussi Ampuja: Lähdekoodiplagiarismin automaattisesta tunnistamisesta
Pro gradu -tutkielma, 41 sivua
Toukokuu 2015

Lähdekoodin plagiointi oppilaitoksissa on ongelma siinä missä luonnollisen kielenkin. Ohjelmointitehtävien käsin tarkastaminen plagioinnin varalta on epäkäytännöllisen aikaavievää ja tehotonta, joten on olemassa tarve lähdekoodiplagiaattien koneelliselle seulonnalle. Koska lähdekoodi on luonteeltaan hyvin erilaista kuin luonnolliset kielet, tähän kuitenkin vaaditaan erityisesti tarkoitukseen suunniteltuja työkaluja. Tässä tutkielmassa luodaan kirjallisuuskatsaus lähdekoodin plagioinnin automaattisessa tunnistuksessa käytettyihin menetelmiin ja ohjelmistoihin.

Avainsanat ja –sanonnat: plagiarismi, lähdekoodi, plagiaatintunnistus

Sisällys

1. Johdanto 1
2. Lähdekoodiplagiarismin peruskäsitteitä 3
   2.1. Lähdekoodi, ja miksi sitä plagioidaan 3
   2.2. Miten lähdekoodia plagioidaan, tai: ”Miksi plagiaatintunnistus on vaikeaa?” .. 4
   2.3. Mikä oikeastaan on lähdekoodiplagiarismia? 6
3. Miten plagiaatintunnistus toimii? 8
   3.1. Ominaisuuksien laskenta 10
   3.2. Rakenteellinen vertailu 12
      3.2.1. Tokenisaatiovaihe 12
      3.2.2. Sekvenssien samankaltaisuuden tarkastelu 14
      3.2.3. N-grammit 17
   3.3. Abstraktit syntaksipuut 19
   3.4. Riippuvuusgraafit 20
   3.5. Latentti semanttinen analyysi 23
   3.6. Tulosten esittäminen käyttäjälle 24
4. Nykyisiä työkaluja ja niiden testausta 27
   4.1. Työkaluja plagioinnin tunnistukseen 27
   4.2. Työkalujen suorituskyvyn arviointia 28
      4.2.1. Käytetty data 29
      4.2.2. Tulokset 35
5. Automaattisen tunnistuksen huijaaminen 38
6. Yhteenveto 40
Viiteluettelo 42

Liitteet

1. Johdanto

Plagiointi tarkoittaa jonkun toisen tekemän työn tai ajatusten esittämistä ominaan ilman asianmukaisia lähdeviitteitä. Yritysmaailmassa ohjelmakoodia plagioivan osapuolen tarkoitus saattaa olla saavuttaa taloudellista hyötyä, ja äärimmäisissä tapauksissa on kyse tekijänoikeusrikoksesta – esimerkkinä vaikkapa avoimen lähdekoodin projektin kopioiminen ja myyminen omana tuotteena. Korkeakouluissa opiskelijat yrittävät plagioimalla läpäistä kursseja, joista eivät joko muuten suoriutuisi tai joihin eivät halua sijoittaa aikaansa ja vaivaansa. Tällöin plagiointi ei riko lakia, mutta on kuitenkin eettisesti arveluttavaa, vastoin hyviä tieteellisiä käytäntöjä ja oppilaitosten sääntöjä.

Akateemisen plagioinnin voidaan ajatella laskevan koko koulutuksen laatua ja murentavan pohjaa sekä sen uskottavuudelta että hyödyllisyydeltä. Internetin sisällön kasvamisen myötä kaikenlainen epärehellisyys on helpompaa kuin aikaisemmin. Suurehko määrä tietoa on kenen tahansa vapaasti varastettavissa, eikä ole realistista ajatella, että opinnäytteitä plagioinnin varalta käsin tarkastamalla saavutettaisiin kovin hyviä saati kustannustehokkaita tuloksia. Useat Suomen korkeakouluista ovatkin viimeisen viiden vuoden aikana ottaneet käyttöön jonkin automaattisen plagiaatintunnistusjärjestelmän. Suosittuja valintoja tuntuvat olevan Urkund ja Turnitin. Kumpikaan näistä ei ilmeisesti kuitenkaan osaa tarkastaa lähdekooditiedostoja siinä mielessä, että ne ottaisivat huomioon niihin liittyvät erityispiirteet.

Lähdekoodiplagiarismin tehokas automaattinen tunnistaminen vaatii hyvin erilaiset menetelmät kuin luonnollinen kieli, yleensä täysin erillisen, nimenomaan tätä tarkoitusta varten suunnitellun ohjelman. Näitä on saatavilla useita, mutta toisin kuin edellä mainitut Urkund tai Turnitin, lähdekoodiplagiarismin tunnistukseen suunnitellut ratkaisut ovat pääsääntöisesti ”ohjelmia” tai ”työkaluja”. Nämä termit kuvaavat niitä paljon tarkemmin kuin ”järjestelmä” tai ”ohjelmisto”, joihin sanana liittyy jonkinlainen implikaatio suuruudesta ja kompleksisuudesta. Melkein kaikki kirjallisuudesta löytyvät ohjelmat ovat yhden tai kahden ihmisen työn tuloksia, joita joko jaetaan ilmaiseksi, tai sitten ne ovat vain tietyn oppilaitoksen sisäisesti kehittämiä ja käyttämiä.

Ohjelmakoodi ei ole ainoa ohjelman osatekijä, jota on mahdollista plagioida. Luvattomasti voi kopioida myös esimerkiksi dokumentaatiota, sisältöä, testidataa tai käyttöliittymäratkaisuja. Tässä tutkielmassa keskityn kuitenkin vain lähdekoodiplagiarismin tunnistukseen.

Aluksi käsitttellään plagiarismia ja sen syitä yleisesti, tarkastellaan lähdekoodiplagiarismin määritelmää sekä kuvataan lähdekoodiin liittyviä erityishaasteita automaattisen tunnistuksen kannalta. Kolmannessa luvussa esitellään erilaisia lähdekoodiplagiarismin tunnistuksessä käytettyjä menetelmiä ja yritetään hieman valottaa niiden toimintaperiaatteita. Neljännessä luvussa luodaan katsaus muutamaan konkreettiseen tarkoitusta varten suunniteltuun työkaluun ja tutkitaan suppealla kokeella, miten hyvin ne tunnistavat eriasteisesti naamioitua plagiointia Java-kielisistä lähdekooditiedostoista. Lopuksi tämän perusteella listataan sellaisia lähdekoodiin tehtävissä olevia

muutoksia, joilla plagiaatintunnistustyökaluja vaikuttaisi olevan mahdollisimman tehokasta huijata, ja toisaalta joiden osalta työkaluja olisi varaa parantaa.

2. Lähdekoodiplagiarismin peruskäsitteitä

2.1. Lähdekoodi, ja miksi sitä plagioidaan

”Lähdekoodilla”, ”ohjelmakoodilla” tai vain lyhyesti ”koodilla” tarkoitetaan jonkin ohjelmointikielen kieliopin mukaista tekstiä, josta joko kääntäjä tai tulkkiohjelma muodostaa sarjan tietokoneen suoritettavissa olevia, konekielistä komentoja. Oletettavasti kaikkiin ohjelmistoalan korkeakoulututkintoihin sisältyy opintojaksoja, joilla opiskelijoiden on tarkoitus osoittaa osaamisensa tuottamalla lähdekoodia. Ohjelmia on mahdollista kopioida luvattomasti siinä missä mitä tahansa muutakin dataa.

Yleisesti plagioinnille on varmasti monta mahdollista tekijää: kiire, epärealistiset tavoitteet kurssin arvosanojen suhteen, vertaispaine yms. Plagiointi voi olla myös tahatonta: joskus oppilaat tekevät keskenään yhteistyötä hieman yli sallittujen rajojen tai eivät täysin hallitse viittauskäytäntöjä. Omaksutut ajatukset voivat alkaa tuntua omilta tai yleisesti tiedossa olevilta – äskettäin luettua lähdekirjallisuutta saattaa kopioida lähes sanasta sanaan huomaamattaankin. Ohjelmakoodin plagiointiin on myös toisenlaisia syitä. Opiskelijat saattavat kokea ohjelmointikurssit hyvinkin haastaviksi; aloittelevan ohjelmoijan voi olla vaikea omaksua tarvittavia ajattelumalleja tai paikantaa tekemiään virheitä. Tämä saattaa johtaa turhautumiseen ja sitä kautta kiusaukseen käyttää vilppiä. Tehtävien tarkastus jää helposti pintapuoliseksi, koska kurssikoot ovat usein suuria, varsinkin opiskelijoille pakollisilla peruskursseilla (esimerkiksi TTY:n perusohjelmointikursseille osallistuu 600-700 opiskelijaa vuosittain [Ahola, 2014]). Juuri näillä kursseilla tarkastus olisi kuitenkin tärkeää, koska niillä plagioinnin esiintyvyystaajuus on suurempaa kuin syventävillä opintojaksoilla [Culwin *et al.*, 2001].

Löfström ja Kupila [2012] toteavat opiskelijoille ja opettajille teettämänsä kyselyn perusteella, että opettajat arvioivat plagioinnin syiksi useimmiten puutteelliset taidot tieteellisen tekstin tuottamisessa, sekä rikkeen vakavuuden aliarvioinnin. Opiskelijat itse sen sijaan kokevat suurimpana motivaattorina tällaiselle toiminnalle sen, että kiinnijäämisen riski on pieni ja mahdolliset sanktiot vähäisiä suhteessa vilpillä saatavaan hyötyyn. Jos kiinnijäämisen kuviteltu riski olisi suurempi, olisiko opiskelijoiden taipuvaisuus plagiointiin pienempi, eli voisiko pelkkä tieto siitä, että oppilaitos käyttää automaattista plagiaatintunnistusohjelmaa, vähentää vilppitapauksia?

Stapleton [2012] on tutkinut tätä vertailemalla plagioinnin esiintymistä kahdessa 22 opiskelijan ryhmässä, joille annettiin joukko esseemuotoisia tehtäviä. Toinen ryhmä tiesi tuotostensa automaattisesta tarkistamisesta ja toinen ei. Tuloksista käy ilmi, että jälkimmäisen ryhmän opiskelijoiden taipuvaisuus plagiointiin – tai ainakin siitä kiinni jäämiseen – oli huomattavasti suurempi. Keskimäärin noin 12% ryhmän palautuksista huomattiin olevan muualta kopioituja. Tarkistuksesta etukäteen tienneiden palauttamista töistä vain neljä prosenttia osoittautui epäilyttäväksi. Tämän perusteella Stapleton toteaa pelkästään järjestelmän käytöllä olevan

jonkinlainen ennaltaehkäisevä vaikutus plagioinnin ilmenemiseen. Samankaltaisen tutkimuksen on tehnyt aiemmin mm. Youmans [2011], mutta hän ei havainnut merkittävää eroa vertailuryhmien välillä.

Käytännön kokemukset automaattisesta plagiaatintunnistuksesta Tampereen teknillisellä yliopistolla vaikuttaisivat tukevan Stapletonin tuloksia. Heti Plakki-järjestelmän käyttöönoton yhteydessä paljastui 70 vilppitapausta, mutta ne vähenivät nopeasti muutamaan kappaleeseen vuodessa, oletettavasti koska tieto järjestelmän käytöstä ja kiinnijäämisen mahdollisuudesta levisi opiskelijoiden keskuudessa. [Ahola, 2014] Vaikka tapausten väheneminen olisikin osittain johtunut siitä, että opiskelijat oppivat plagioimaan paremmin ja varovaisemmin, voidaan tämä silti nähdä hyvin positiivisena tuloksena.

Useimmat Suomen korkeakoulut ilmoittavat käyttävänsä jonkinlaista automaattista prosessia plagiarismin etsimiseen. Vaikuttaisi siltä, että niillä kuitenkin tarkastetaan lähinnä väitöskirjoja tai muita suuria yksittäisiä opintosuorituksia. Lähdekoodin luvatonta kopiointia ei ilmeisesti kovin systemaattisesti valvota, tai ainakaan siitä ei laajalti tiedoteta. Esimerkiksi Surakka ja muut [2003] toteavat ohjelmakoodiplagiarismin kuitenkin olevan jonkinasteinen ongelma myös Suomessa. Tampereen teknillisessä yliopistossa [Järvinen, 2014] ja Aalto-yliopistossa [Ahtikainen *et al.*, 2006] on käytössä talon sisäisesti kehitettyjä lähdekoodiplagiarismin seulontaan tarkoitettuja ohjelmia, mutta nämä tuntuvat olevan enemmänkin poikkeuksia sääntöön.

Sähköistä plagiaatintunnistusta kohtaan on esitetty myös kritiikkiä. Hyvin usein siinä on mukana eettisiä näkökulmia: osa järjestelmistä esimerkiksi toimii ainoastaan Software as a Service -tyyppisinä ”pilvipalveluina”, jotka vaativat, että tarkastettavat tehtävät lähetetään ulkopuoliselle palvelimelle analyysiä varten. Mahdollisesti ne jopa tallennetaan järjestelmän tietokantaan lähdeaineistoksi tuleville tarkastuksille. Oleellisesti kuitenkin palvelun käyttäjä menettää kontrollinsa siihen, mitä sinne ladatuille tiedostoille lopulta tapahtuu, mikä puolestaan on vastoin monien oppilaitosten käytäntöjä, joiden mukaan oppilaiden tuotoksia ei saisi luovuttaa kolmansille osapuolille.

Plagioinnintunnistusjärjestelmien antamia tuloksia ei voida yksistään pitää todisteena vilppiin syyllistymisestä. Epäilyttävät työt on aina tarkistettava käsin ja sitten harkittava, onko kyseessä ehkä väärä positiivinen vai aidosti tutkimisen arvoinen yhtäläisyys. Järjestelmien rooli on lähinnä merkata mahdollisesti epäilyttävät ohjelmakoodiparit ja osoittaa niiden väliset samankaltaisuudet. Vasta tästä alkaa varsinainen todisteiden keruu, jonka jälkeen olisi päätettävä, onko aihetta jatkotoimenpiteisiin. Tutkimuseettinen neuvottelukunta [2012] antaa yleisiä ohjeita siitä, miten akateemista epärehellisyyttä tulisi korkeakouluissa käsitellä.

2.2. Miten lähdekoodia plagioidaan, tai: ”Miksi plagiaatintunnistus on vaikeaa?”

Jos plagiointi on tahallista ja tietoista, plagioija oletettavasti yrittää naamioida sen jotenkin. Valitettavasti lähdekoodin tapauksessa tämä on helppoa. Ohjelmaan on yksinkertaisilla etsi ja korvaa

-menetelmillä mahdollista saada aikaan muutoksia, joiden jälkeen sillä on silmämääräisesti arvioiden hyvin vähän yhteistä alkuperäisen kanssa. Esimerkki tästä on kuvassa 1.

Lähdekoodin plagiointia voidaan yrittää peittää joko *leksikaalisilla* tai *rakenteellisilla* muutoksilla [Joy & Luck 1999]. Leksikaalinen eli kielillinen muutos tarkoittaa tunnisteiden uudelleennimeämistä tai muuta pinnallista naamiointia, joka ei vaikuta mitenkään siihen miten kääntäjä ohjelmakoodin tulkitsee. Rakenteellinen muutos taas voisi olla esimerkiksi if-else-haarojen runkojen keskenään vaihtaminen ja ehtolauseen kääntäminen negaatiolla. Whale [1990] listaa usein kirjallisuudessa siteeratun joukon eriasteisia naamioimismenetelmiä, alkaen helpoimmasta toteuttaa ja edeten nousevassa ”hienostuneisuusjärjestyksessä”:

- Kommenttien ja koodin tyylin muokkaus
- Tunnisteiden nimien muokkaus
- Operandien järjestyksen vaihtaminen lausekkeissa
- Muttujien tietotyyppien vaihtaminen
- Ehtolauseiden korvaaminen loogisesti vastaavilla
- Tarpeettoman koodin lisääminen
- Toisistaan riippumattomien koodisekvenssien uudelleenjärjestely
- Silmukkarakenteiden muokkaus
- Ehtorakenteiden muokkaus
- Proseduurikutsun korvaaminen proseduurin rungolla
- Plagioidun ja oman koodin yhdisteleminen.

Näistä kaksi ensimmäistä edustaa leksikaalisia, kosmeettisia muutoksia. Niiden aikaansaaminen on triviaalia, mutta toisaalta myös niiden automaattinen tunnistaminen on (kuten myöhemmin huomaamme) suhteellisen helppoa. Loput, ohjelman rakenteeseen vaikuttavat muutokset, on huomattavasti vaikeampi havaita. Toisaalta ne vaativat tekijältään vaivannäköä ja osaamista, jonka osoittaminen oli alun perinkin ohjelmointitehtävän tarkoitus. Siispä on kyseenalaista, miksi tällaiseen kykenevä opiskelija vaivautuisi plagioimaan, tai onko ylipäätään kovin tärkeää, että automaatiset järjestelmät kykenisivät tunnistamaan tällaiset plagiaatit. Täytyy myös harkita, missä kohtaa laajoja rakenteellisia muutoksia läpikäynyt ohjelma lakkaa olemasta plagiaatti.

```
boolean onkoAlkuluku (int luku)
{
    if (luku < 2)
        return false;

    int jakaja = luku - 1;
    while (jakaja > 1)
    {
        if (luku % jakaja == 0)
            return false;

        jakaja--;
    }

    return true;
}
```

```
boolean isPrime(int n) {
    if (2 >= n) { return false; }
    for (int m = n - 1; m > 1; m--) {
        if (n % m == 0) { return false; }
    }
    return true;
}
```

Kuva 1 - Lähdekoodi ja siitä tuotettu, suhteellisen yksinkertaisilla keinoilla naamioitu plagiaatti

2.3. Mikä oikeastaan on lähdekoodiplagiarismia?

Parker ja Hamblen [1989] määrittelevät ohjelmaplagiaatin ”ohjelmaksi, joka on tuotettu toisen ohjelman pohjalta pienillä rutiinimuokkauksilla”. Määritelmä on enemmänkin realistinen kuin ideaali: varmasti myös keskikokoiset rutiinimuokkaukset ovat plagiarismia, mutta niitä on niin vaikea tunnistaa, että ne on helpompi määritellä pois koko ongelma-alueesta. Cosma ja Joy [2012] tekivät kyselyn, johon vastasi 120 ohjelmointia korkeakoulussa parhaillaan opettavaa tai joskus opettanutta henkilöä. Kysymyksissä kuvailtiin erilaisia skenaarioita, ja vastaajien oli tarkoitus kertoa, onko kyseessä heidän mielestään plagiointitapaus vai ei. Tuloksista käy ilmi, että kaikki vastaajat kannattivat ”nollatoleranssia” plagiarismin suhteen, mutta olivat kuitenkin paikoin eri mieltä siitä, mikä oikeastaan on plagiarismia. Tämä korostui varsinkin kysymyksissä, jotka koskivat opiskelijoiden välistä yhteistyötä, koodin uudelleenkäyttöä ja muualta hankitun koodin alkuperään viittaamista.

Ajatusten jakamisen ja toteutuksen kopioinnin välillä ei ole kovin selkeää rajaa. Jälkimmäinen on selvästi kiellettyä, mutta edellinen pääsääntöisesti sallitaan, ja siihen jopa kannustetaan. Voi olla pedagogisesti hyödyllistä, että opiskelijat keskustelevat keskenään ohjelmointitehtävistä ja niiden ratkaisuista [Cosma & Joy 2012]. Jos tässä keskustelussa kuitenkin mennään tarpeeksi matalalle abstraktiotasolle, on kyse jo toteutuksen jakamisesta, mikä taas voidaan nähdä plagiarismina.

Varmastikin plagiarismin määritelmä elää hieman myös tehtävänannon mukana. Opintojen alkuvaiheessa, ohjelmoinnin perusteita opeteltaessa opiskelijoiden tyypillisesti odotetaan tuottavan ratkaisunsa täysin itse. Myöhemmin, kun tehtävät ovat laajempia, voi olla hyväksyttävää ja suotavaa etsiä ohjelman osaongelmiin valmiita ratkaisuja ja muokata niitä omiin käyttötarkoituksiin sopiviksi. Tällöinkin toki olisi hyvä viitata lähdekoodin alkuperään vaikka kommentissa. Koodin uudelleenkäyttö ja periaate, että "pyörää ei kannata keksiä uudelleen", nähdään pääsääntöisesti hyvinä asioina, mutta ne on paikoin vaikea sovittaa yhteen plagiarismin perinteisen määritelmän kanssa. "Alihankintaplagiointi", eli se, että joku muu kirjoittaa opiskelijan työn tämän puolesta, on todellinen, mutta automaattiselle tunnistusjärjestelmälle mahdoton ongelma, joten siihen ei tässä puututa.

3. Miten plagiaatintunnistus toimii?

Jotta koneellinen plagiaatintunnistus on mahdollista, on kysymys ”Onko ohjelma X plagiaatti?” esitettävä muodossa ”Onko ohjelma X epäilyttävän samankaltainen jonkin toisen, ohjelman Y kanssa?”. Tunnistuksen tehtäväksi jää verrata sille annettuja ohjelmia referenssijoukkoon, muodostaa jonkinlainen mielipide epäilyttävistä lähdekoodipareista, ja lopuksi esittää vertailun tulokset käyttäjälle mielekkäässä muodossa. On huomattava, että ohjelmallisesti voidaan tutkia ainoastaan lähdekoodien objektiivista samankaltaisuutta, ei syitä siihen, eikä sitä, onko kyse plagioinnista vai ei.

Koska kysymys on lopulta ohjelmien samankaltaisuuden vertailusta, plagiaatintunnistus on osittain päällekkäinen ongelma esimerkiksi toistetun koodin etsimisen kanssa. Jälkimmäinen on tarpeellista usein sellaisten järjestelmien tapauksessa, joilla on suuri koodipohja. Silloin monet ohjelman osat saattavat tehdä saman asian. Ohjelmiston ylläpidettävyyden ja muokattavuuden kannalta nämä kohdat olisi tietenkin tärkeää tunnistaa ja mahdollisesti poistaa, koska ne voivat sisältää esimerkiksi jaetun virheen. Vaikka varsinaista virhettä ei olisikaan, muutokset tiettyyn toiminnallisuuteen pitää silti ohjelmoida useampaan paikkaan kuin olisi tarpeellista. Toistetun koodin etsimiseen tarkoitetut menetelmät on usein suunniteltu käsittelemään vain yhtä ohjelmaa, ja etsimään siitä sisäisiä samankaltaisuuksia. Monet niistä soveltuvat kuitenkin hyvin myös plagiaatintunnistukseen (ja päinvastoin).

Referenssimateriaali, eli niiden lähdekoodidokumenttien joukko, joihin käsiteltäviä palautuksia vertaillaan, ei tietenkään voi olla yhtäsuuri kuin kaiken kirjoitetun lähdekoodin joukko. Luonnollisen kielen tapauksessa referenssimateriaalin valinta on tärkeää; melkein aiheesta kuin aiheesta on mahdollista löytää relevanttia tekstiä uudelleenkäytettäväksi. Lähdekoodin leikkaa-liimaa-plagiointi julkisista lähteistä on kuitenkin onneksemme vaikeaa. Tehtävänannot ovat pääsääntöisesti hyvin tarkasti määriteltyjä, eikä vaatimukset täyttävää valmista ohjelmaa löytyne suoraan internetin hakukoneella. Ohjelmakoodin kopioiminen rajoittuukin pääasiassa vertaisplagiointiin, eli opiskelijat kopioivat muiden samaa tehtävää ratkaisseiden koodia [Sraka & Kaucic, 2009]. Referenssimateriaali saattaa siis olla järkevää rajoittaa kattamaan pelkästään kurssilaisten (ja mahdollisesti vaikkapa aikaisempina vuosina samaa tehtävää suorittaneiden) ratkaisut.

Koska ohjelmointikielet ovat muodollisia, niiden sisältämä informaatio on luonnollisia kieliä helpompi käsitellä koneellisesti – sitä vartenhan ne on suunniteltukin. Toisaalta niiden kielioppi on hyvin rajattu: C-kieli sisältää alle sata varattua sanaa (vrt. luonnollisten kielten kymmeniin, joskus satoihin tuhansiin). Samat avainsanat esiintyvät usein yhdessä. Jopa muuttujat saavat helposti samoja nimiä eri ratkaisuissa (x, line, size, min, max, apu, tmp jne). Tämä yhdistettynä siihen, että kaikki vertailussa olevat ohjelmat yrittävät ratkaista identtistä tehtävää, luo perustavanlaatuisen ongelman: ratkaisut ovat väistämättä samankaltaisia.

Liian hienostuneet tunnistusmenetelmät (sellaiset, jotka huomaavat hyvin naamioidut plagiointiyritykset) saattavat olla epäytännöllisiä, koska niillä voi olla tapana olla liian ”leveitä”, eli

löytää plagiointia sieltäkin, missä sitä ei ole. Jonkinlainen kompromissi on siis tehtävä liian suuren manuaalisen tarkastustyön ja satunnaisen plagioinnista rangaistuksetta selviämisen välillä. Tunnistuksen tarkkuus ja automaattisen tarkastuksen käyttökelpoisuus on suoraan verrannollista siihen, kuinka monella tapaa kulloinenkin tehtävänanto on mahdollista ratkaista. Kuten Whale [1990] toteaa: on olemassa alaraja sille, kuinka monimutkaisissa tehtävissä automaattinen plagiaatintunnistus on järkevää.

Usein käytetty tunnistuksen hyvyyden arviointiin liittyvä termipari on saanti (*recall*) ja tarkkuus (*precision*). Saannilla tarkoitetaan vertailujoukosta oikein tunnistettujen plagiointitapausten määrän suhdetta kaikkiin siinä esiintyviin plagiointitapauksiin. Tarkkuus on oikein tunnistettujen tapausten määrä suhteessa kaikkiin tunnistettuihin tapauksiin. Saannin paraneminen johtaa usein tarkkuuden huononemiseen.

Tyypillisesti vertailualgoritmit käsittelevät kerrallaan yhtä ohjelmakoodiparia. Koska jokaista työtä joudutaan vertaamaan jokaiseen toiseen, on vertailualgoritmin aikakompleksisuus luonnostaan lähellä luokkaa $O(n^2)$. Tarvittavien vertailujen määrä saadaan periaatteessa kaavalla

$$^1/_2\, n(n-1) + nm\ ,$$

missä n on tarkastettavien töiden määrä, ja m sellaisten referenssijoukon alkoiden lukumäärä, jotka eivät itse ole tarkastuksen kohteena. Käytännössä monet luvussa 4 käsitellyistä ohjelmista tuntuvat kuitenkin vertaavan jokaista ohjelmaparia kahdesti (molempiin suuntiin), eli tekevän n^2-n vertailua. Vaikka laskenta-aika ei varsinaisesti olekaan kriittisessä roolissa (tarkastajalle on todennäköisesti paljon tärkeämpää, että tulokset ovat hyödyllisiä ja luotettavia), kasvavat suoritusajat helposti epäkäytännöllisiin mittasuhteisiin. Hyvänä esimerkkinä tästä toimii esimerkiksi TTY:n Plakki-ohjelma, jonka versio 1.0 käsitteli Järvisen [2014] mukaan 12672 lähdekooditiedoston joukkoa lähes 15 vuorokautta, ja sai siinä ajassa vasta yhden vaiheen tunnistuksesta valmiiksi. Tässäkin on siis tehtävä kompromissejä, tai vähintään otettava asia huomioon.

Ehkäpä suoraviivaisin lähestymistapa olisi etsiä yhteisiä tekstisegmenttejä vertailtavien dokumenttien välillä ja todeta dokumentit epäilyttäviksi, jos näitä esiintyy paljon tai ne ovat huomattavan pitkiä. Oleellisesti siis samaan tapaan kuin yksinkertainen luonnollisia kieliä käsittelevä plagiaatintunnistin tekisi. Yhteisten tekstikatkelmien etsiminen pitkistä dokumenteista on kuitenkin laskennallisesti raskasta, ja ongelma korostuu entisestään referenssimateriaalin ollessa suuri. Toisaalta puhdas tekstuaalinen vertailu on kieliriippumatonta; järjestelmän ei tarvitse ymmärtää yhtään mitään ohjelmointikielestä, jolla vertailtavat ratkaisut on kirjoitettu. Vertailijaa ei tarvitse muokata, vaikka opetuskieli vaihtuisi toiseen. Yksinkertainen koodin leikkaa-leimaa-plagiointi voidaan tunnistaa aivan hyvin näinkin.

Ohjelmakoodi on kuitenkin huono kohde tällaiselle analyysille. Taitamattomankin ohjelmoijan on helppo tehdä koodiin yksinkertaisia muutoksia, jotka kuitenkin muuttavat sen tekstimuotoista esitystä radikaalisti. Tämä saadaan aikaan pelkästään kommentteja, rivinvaihtoja ja sulkumerkkejä

muokkaamalla sekä tunnisteet uudelleennimeämällä.Yleisesti ottaen parempi vaihtoehto onkin ennen vertailua etsiä ohjelmakoodeille sellainen esitysmuoto, jota plagioijan on vaikeampi muuttaa.

3.1. Ominaisuuksien laskenta

Ominaisuuksien laskenta (engl. *attribute-counting* tai *attribute-metric system*) on varhainen menetelmä, jossa lasketaan joidenkin ominaispiirteiden esiintymistä lähdekoodissa. Taustalla on oletus, että näin saatavien tunnuslukujen ollessa samankaltaisia kahdessa vertailtavassa dokumentissa myös dokumentit itsessään ovat samankaltaiset.

Ensimmäisiä ajatuksia koneellisesta plagiaatintunnistuksesta julkaistiin 1970-luvulla. Halstead [1977] yritti löytää algoritmeista konkreettisia, mitattavia ominaisuuksia ja suhteita niiden välillä, jotta esimerkiksi ohjelman kompleksisuutta tai siinä esiintyvien virheiden määrää voitaisiin karkean arvioinnin sijasta muodollisemmin approksimoida. Hänen työnsä tuloksena syntyivät nk. ”Halsteadin metriikat”. Ottenstein [1976] sovelsi käytäntöön ajatusta, että näillä mittareilla voitaisiin arvioida myös ohjelmien samankaltaisuutta, ja kirjoitti ilmeisesti ensimmäisen automaattisen plagiaatintunnistimen. Ottensteinin ohjelma osasi etsiä samankaltaisuuksia ainoastaan FORTRAN-kielisistä lähdekoodidokumenteista, mutta hän arveli, että tulevaisuudessa plagiaatteja voitaisiin tunnistaa koneellisesti paitsi muilla ohjelmointikielillä kirjoitetusta tekstistä, myös luonnollisesta kielestä.

Ottensteinin järjestelmä käytti Halsteadin ohjelmistometriikoden neljää perussuuretta, ja näin ollen laski lähdekoodille neljä tunnuslukua:

n_1 - operaattoreiden kokonaismäärän

n_2 - operandien kokonaismäärän

N_1 - uniikkien operaattoreiden määrän

N_2 - uniikkien operandien määrän.

Jokaiselle palautukselle muodostuu näin nelikomponenttinen ”sormenjälki”, $P = \{ n_1, n_2, N_1, N_2 \}$. Eri ohjelmakoodeista laskettuja sormenjälkiä vertailtiin keskenään, ja mikäli jokin tarkasteltava ohjelmapari sisälsi identtiset arvot kaikissa, Ottensteinin järjestelmä merkkasi sen epäilyttäväksi.

Myöhemmin kävi selväksi, että tällä tavoin ei saavuteta riittävän hyviä tuloksia [Berghel & Sallach 1984; Whale 1990]. Sittemmin menetelmää on pyritty hienosäätämään, esimerkiksi vaihtamalla laskettavia ominaisuuksia, tai painottamalla niitä eri tavoin kun lopullista samankaltaisuuslukemaa lasketaan. Esimerkiksi hieman uudempi, Pascal-kielistä lähdekoodia analysoiva ohjelmisto Accuse [Grier, 1981] käyttää Ottensteinin valitsemien tunnuslukujen lisäksi kolmea muuta mittaria: koodirivien määrää, esiteltyjen (ja käytettyjen) muuttujien määrää sekä kontrollirakenteiden määrää. Lisäksi Grierin ratkaisussa eri tunnusluvuille on asetettu ”kiinnostavuusikkuna” ja ”tärkeys”. Jos kahden eri dokumentin jonkin tunnusluvun erotuksen itseisarvo alitti ikkunan koon, kasvatettin dokumenttien samankaltaisuudeksi ilmoitettavaa

lukua sen mukaan, mikä oli kyseisen tunnusluvun tärkeys ja paljonko dokumenttien arvot toisistaan erosivat.

Pitkälle 1980-luvulle asti suurin osa lähdekoodiplagiarismia käsittelevästä tutkimuksesta vaikuttaa keskittyneen lähes yksinomaan parempien tunnuslukujen löytämiseen. Whalen [1990] mukaan ollakseen käyttökelpoinen, tunnusluvun tulee:

- mitata sellaista ohjelman sisäsyntyistä ominaisuutta, jota on vaikea vaihtaa naamiointitarkoituksessa
- olla “luja”, eli vaihdella vain minimaalisesti pinnallisista muokkauksista ohjelmaan
- olla suhteellisen helposti vertailtavissa
- päteä laajaan kirjoon ohjelmointikieliä.

Lukuisia eri variaatioita ominaisuuksien laskennasta on esitelty ja käytetty vaihtelevalla menestyksellä. Edellämainittujen lisäksi huomionarvoinen on ilmeisesti ainakin Faidhin ja Robinsonin [1987] menetelmä.

Verco ja Wise [1996] vertailevat ominaisuuksien laskentaa uudempiin ”rakenteellisiin” menetelmiin. Heidän mukaansa (päinvastoin kuin ensin yleisesti arveltiin), ominaisuuksien laskennalla on taipumus löytää olematontakin plagiointia. Toisaalta väärät positiiviset on helppo havaita manuaalisessa tarkastusvaiheessa: eiväthän plagiaattikandidaatit todennäköisesti muistuta ollenkaan toisiaan, muuten kuin siinä mielessä, että niissä sattuu olemaan sama määrä asioita. Ominaisuuksien laskenta tunnistaa hyvin plagioimisyritykset, jossa merkittävä osa lähdekoodista on kopioitu. Toisaalta jos koodi on plagioitu vain osittain, useammasta lähteestä, tai siihen on lisätty jotain, ei mikään esitelty tunnuslukujoukko pysy muuttumattomana.

Kaikenlaisen ”syntaktisen sokerin” eli ohjelmoijan työtä helpottavien, tulkkiin tai kääntäjään upotettujen oikoteiden (mm. muuttujan inkrementointi ++-notaatiolla) hyödyntäminen vääristää ainakin Halsteadin metriikoiden tuottamia tunnuslukuja huomattavasti. Lähdekoodit voitaisiin ehkä yrittää normalisoida esiprosessoimalla ne jotenkin, tavoitellen mahdollisimman yhtenäistä esitysmuotoa riippumatta siitä, miten alkuperäinen kirjoittaja on tietyn lausekkeen muotoillut. Tämä kuitenkin lisäisi algoritmin kompleksisuutta, huonontaisi tarkkuutta, eikä ratkaisisi kuin yhden menetelmän ongelmista. Pääsääntöisesti ominaisuuksien laskenta pärjäsi Vercon ja Wisen vertailussa huonommin tai korkeintaan yhtä hyvin kuin seuraavana käsiteltävät, ohjelman rakennetta tarkastelevat menetelmät.

Verrattuna mihin tahansa muuhun tässä tutkielmassa käsiteltyyn lähestymistapaan, ominaisuuksien laskennalla on kuitenkin pelkästään sille ominainen etu: se on hyvin nopeaa. Jokainen tarkastettava työ tarvitsee prosessoida vain kerran, ja kun halutut tunnusluvut on laskettu, niiden keskenään vertailu on laskennallisesti kevyttä. Jos tunnusluvut jostain syystä tarvitsee tallettaa pysyvästi, on niiden tarvitsema massamuistimääräkin hyvin vaatimaton.

3.2. Rakenteellinen vertailu

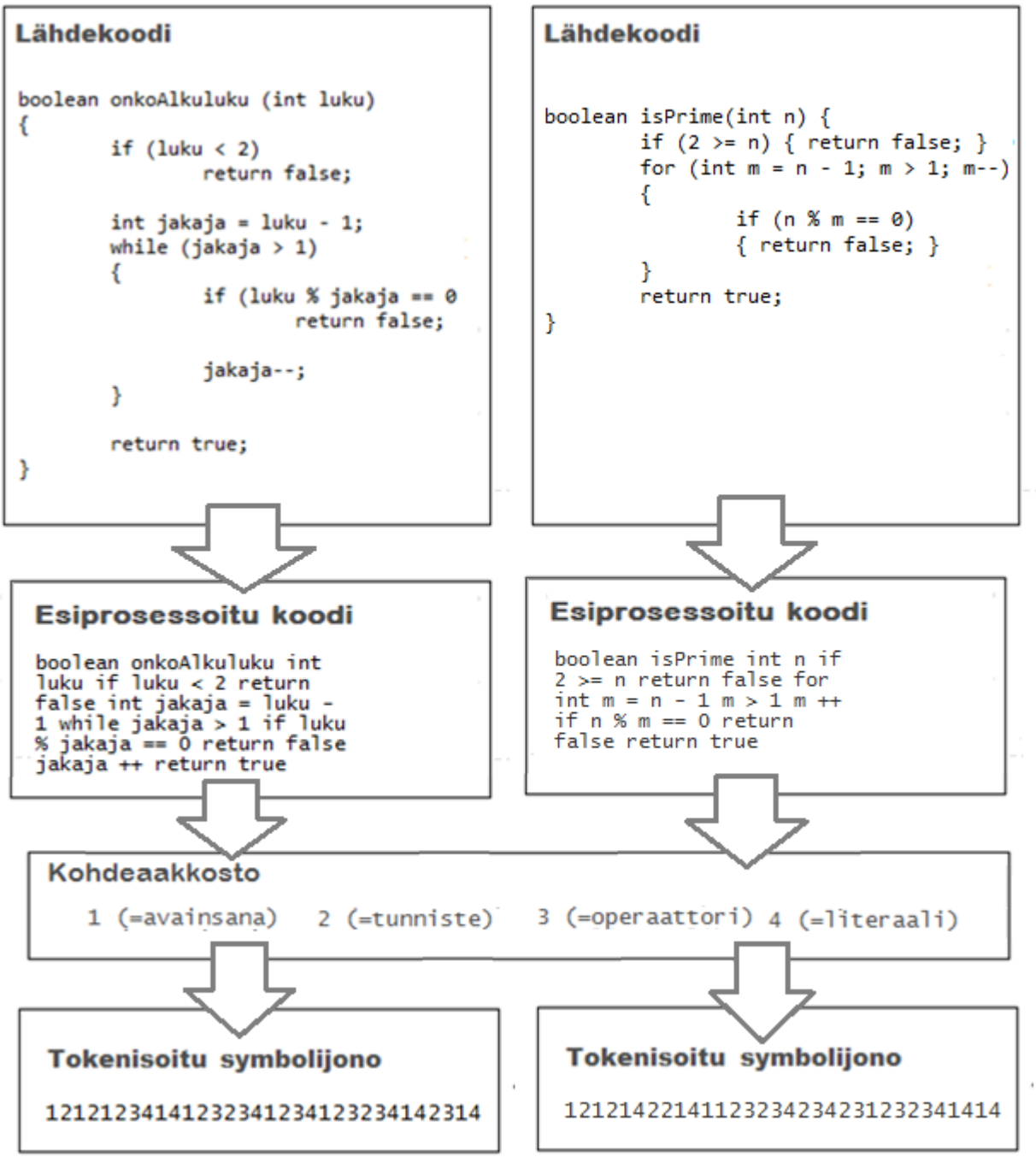
Rakenteellinen vertailu (engl. *structure-metrics*) pyrkii – nimensä mukaisesti – tarkastelemaan lähdekoodin rakennetta, tai ehkä paremminkin siinä esiintyvien elementtien järjestystä, sen sijaan että vain laskisi niiden esiintymistaajuuksia. Vaikka eri implementaatiot eroavat toisistaan huomattavasti, prosessi voidaan yksinkertaistaen jakaa kolmeen vaiheeseen [Zeidman 2004]:

1. Poistetaan koodista kommentit, ylimääräiset välilyönnit, mahdollisesti sulkeet yms, ylipäätään kaikki minkä ei haluta vaikuttavan tulokseen.
2. ”Tokenisoidaan” lähdekoodi, eli muutetaan se jonoksi symboleita (engl. *token*) jonkinlaisten muuntotaulukon mukaan. Avainsanat, operaattorit, muuttujat ja muut koodin termit korvataan niitä vastaavilla symboleilla.
3. Vertaillaan näin saatuja symbolijonoja keskenään.

Ensimmäisessä vaiheessa pyritään tekemään turhaksi kaikkein alkeellisimmat naamiointimenetelmät, kuten kommenttien sekä koodin yleisen tyylin muokkaamisen. Toisen vaiheen funktio on varmistaa, että esimerkiksi muuttujien uudelleennimeäminen ei vaikuta tulokseen, sekä toisaalta nopeuttaa kolmatta vaihetta, jossa varsinainen vertailu tapahtuu. Kahdessa ensimmäisessä vaiheessa kuitenkin hukataan paljon informaatiota: samalla tavalla väärinkirjoitettu muuttujan nimi tai muokkaamattomana kopioitu kommentti voitaisiin nähdä vahvana näyttönä plagioinnista [Zeidman 2004]. Saattaa siis olla järkevää tallettaa myös tämä informaatio, ja ehkä vertailla sitä erilaisin menetelmin, täysin erillään muusta ohjelmasta.

3.2.1. Tokenisaatiovaihe

Tokenisaatio on kielellisen analyysin menetelmä, jossa teksti pilkotaan jollain tavalla merkityksellisiin palasiin. Meidän tapauksessamme sillä pyritään normalisoimaan lähdekoodia muuttamalla sen esitysmuoto sarjaksi symboleita, jotka vastaavat ohjelmointikielen peruselementtejä, kuten tunnisteita tai avainsanoja. Esimerkiksi lähdekoodirivi *int i = 30 + b*; saattaisi tokenisoitua muotoon: <*avainsana*>,<*tunniste*>,<*operaattori*>,<*literaali*>,<*operaattori*>,<*tunniste*>. Tämä ei ole ainoa vaihtoehto. Mahdollisten symbolien joukko, eli kohdeaakkosto, voi toki olla myös erilainen. Esimerkiksi eri muuttujatyypeille saattaa olla omat symbolinsa kohdeaakkostossa. Tokenisaatiovaiheessa yritetään siis löytää lähdekoodissa esiintyville yksittäisille merkkijonoille jonkinlainen syntaksista irrallinen merkitys, vaikka myöhemmin suoritettava vertailu tutkisikin vain tuotetun symbolisekvenssin rakennetta, eikä ohjelman semantiikkaa sinänsä. Kuvassa 2 on havainnollistettu tokenisaatioprosessia kuvan 1 lähdekoodipätkille.



Kuva 2 - Kuvan 1 lähdekoodeille suoritettu tokenisaatio (käyttäen hyvin suppeaa kohdeaakkostoa)

Tokenisaatio on keskeinen osa myös kääntäjien toimintaa. Niiden täytyy tietenkin ottaa huomioon muun muassa literaalien arvot ja operandien tyypit (koska niillä on huomattava vaikutus siihen, miten ohjelma lopulta toimii). Plagiaatintunnistuksessa tällaiset yksityiskohdat on mahdollista jättää täysin huomiotta, koska tässä yhteydessä tokenisaatiolla pyritään vain muodostamaan varsinaiselle vertailijalle syötteeksi sellainen symbolijono, joka kuvaa ohjelman rakenteesta

oleellisen osan. Kommentit, tunnisteiden nimet, ja sulkumerkit voidaan nähdä samankaltaisuuden arvioinnin kannalta hyödyttömiksi, koska niiden muokkaus on helppoa. Tokenisaatio, koodin normalisoinnin lisäksi, myös nopeuttaa vertailuvaihetta. Tieto kunkin symbolin tyypistä voidaan tallettaa vaikkapa yhteen tavuun pitkän merkkijonon sijaan, jolloin tarvittavan muistin määrä pienenee ja alkioiden yhtäsuuruuden tarkastelu nopeutuu.

Tokenisaatiota varten järjestelmä voi joko täysin jäsentää lähdekoodidokumentin, kuten kääntäjä tai tulkki tekisi, tai sitten käydä sitä sana kerrallaan läpi ja tutkia sanakirjan (eli avain-arvo -pareja sisältävän listan) ja mahdollisesti esimerkiksi säännöllisten lausekkeiden perusteella, mikä symboli tulisi valita kuvaamaan kulloistakin merkkijonoa. Lähdekoodissa esiintyvän sanan konteksti kuitenkin vaikuttaa sen merkitykseen, aivan kuten luonnollisessakin kielessä. Esimerkiksi Javan ’+’-operaattori saattaa merkitä joko yhteenlaskua tai kahden merkkijonon yhdistämistä. Tällaiset erot on pelkän sanakirjan avulla hankalaa tunnistaa oikein. Voidaan kuitenkin ajatella, että tämä ei ole kovin merkittävä ongelma, koska jonkinlainen approksimaatio koodin rakenteesta riittää hyvin. Täydellisen jäsentämisen etuina on lähdekoodissa esiintyvien elementtien oikeellisempi tulkinta ja sen myötä kategorisointi. Toisaalta uuden ohjelmointikielen lisääminen järjestelmään on huomattavan työlästä.

Joissain tapauksissa tokenisaattori saattaa suorittaa myös pidemmälle menevää käsittelyä lähdekoodille. Esimerkiksi funktiokutsut saatetaan korvata niiden rungolla tai silmukat keriä auki [Wise, 1993]. Myös koodilohkoja voidaan järjestää jonkin logiikan mukaan. Kaikissa näissä toimenpiteissä tarkoituksena on tehdä tunnistuksesta vastustuskykyisempi erilaisille huijausyrityksille.

3.2.2. Sekvenssien samankaltaisuuden tarkastelu

Kun symbolijonot tarkasteltaville lähdekoodeille on muodostettu, niitä pitää vielä vertailla keskenään. Periaatteessahan kyse on pelkästä merkkijonovertailusta, mutta kahden sekvenssin samankaltaisuus ei ole mitenkään yksiselitteistä. Ovatko merkkijonot ”aaaa” ja ”baaa” samankaltaiset? Entä ”1122” ja ”2211”? Huomaamme, että tarkka ekvivalenssin tutkiminen ei toimi, vaan sekvenssit pitää todeta epäilyttäviksi jos ne ovat tarpeeksi ”sinnepäin”. Mittareina on käytetty mm. pisimpiä yhteisiä alijonoja tai Levenshteinin etäisyyttä.

Levenshteinin etäisyys (*Levenshtein distance* tai *edit distance*) [Levenshtein, 1966] mittaa kahden sekvenssin eroavaisuutta laskemalla pienintä mahdollista tarvittavien merkkikorvausten, poistojen tai lisäysten määrää, jotta toisesta vertailtavasta sekvenssistä saadaan identtinen toisen kanssa. Tarkastellaan esimerkiksi merkkijonoja S_1=”kassah” ja S_2=”rassi”. Levenshteinin etäisyys, $LD(S_1,S_2)$, on 3, koska S_1:n ensimmäinen ja toiseksi viimeinen merkki eroavat S_2:n vastaavista, ja S_2:sta puuttuu S_1:n lopettava ”h” – tarvitaan siis kaksi korvausta ja yksi lisäys tai poisto. Algoritmin aikakompleksisuus on luokassa $O(nm)$, jossa n ja m ovat vertailtavien sekvenssien pituudet.

Pisin yhteinen alijono (*Longest Common Subsequence*, LCS) on pisin mahdollinen sekvenssi, jonka alkiot löytyvät samassa järjestyksessä kaikista syötesekvensseistä (tai molemmista, koska

vertailijat käsittelevät yleensä yhtä lähdekoodiparia kerrallaan). Alkioiden ei tarvitse olla siinä mielessä peräkkäin, että niiden välissä ei olisi muita alkioita, sama keskinäinen järjestys riittää. Esimerkiksi merkkijonojen ”blahlblah” ja ”dadhdhadh”, LCS on ”ahah”. Yleisessä tapauksessa pisimmän yhteisen alijonon laskeminen on NP-kova ongelma. Onneksi, kuten edellä huomasimme, sitä käytetään tässä tapauksessa vertailemaan tasan kahta sekvenssiä kerrallaan. Tällöin LCS voidaan laskea polynomiaalisessa ajassa [Hirschberg, 1975], koska ongelma on rekursiivisesti jaettavissa aina pienempiin, osittain päällekkäisiin aliongelmiin.

Samankaltaisuutta voi lopulta mitata vaikkapa pisimmän yhteisen alijonon pituudella jaettuna lyhemmän vertailtavan sekvenssin pituudella, tai kaavalla

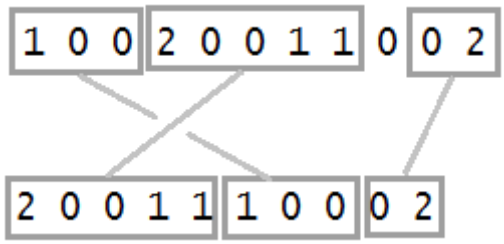
$$1 - \frac{Levenshteinin\ etäisyys}{pidemmän\ sekvenssin\ pituus},$$

jolloin syötesekvenssien samankaltaisuudeksi saadaan yksiselitteinen luku väliltä [0 – 1]. Sekä Levenshteinin etäisyyttä että pisimpiä yhteistä alijonoja on käytetty plagiarismintunnistuksen lisäksi muun muassa oikeinkirjoituksen tarkistuksessa, puheen- ja hahmontunnistuksessa sekä DNA-analyysissä. Varsinkin näistä viimeisellä on paljon yhteistä tietojenkäsittelytieteen ongelmien kanssa. DNA-sekvenssien analysoinnissa käytetty Needlemanin-Wunschin-algoritmi on periaatteiltaan hyvin samankaltainen kuin LCS, mutta se ”rankaisee” tulokseen poimittavien alkioden välisestä etäisyydestä vertailtavissa sekvensseissä. Tällöin sen tuottama tulos saattaa olla lyhyempi kuin LCS, mutta tulokseen valitut alkiot ovat lähempänä toisiaan alkuperäisessä sekvenssissä. [Wise, 1996]

Kaikki edellämainitut ovat vertailumenetelminä kuitenkin varsin kehnoja ottamaan huomioon transpositioita, eli tässä tapauksessa toisistaan riippumattomien koodisekvenssien uudelleenjärjestelyä. Tarkastellaan lähdekooditiedostoa, joka koostuu kahdesta yhtä suuresta, järjestysriippumattomasta koodilohkosta (esimerkiksi kahdesta metodista, tai kahdesta niin suuresta switch-case-haarasta, että ne muodostavan merkittävän osan koko ohjelmasta). Vaihdetaan niiden paikkaa, ja tutkitaan samankaltaisuutta alkuperäiseen. Levenshteinin etäisyyttä laskeva vertailija toteaa eron olevan todennäköisesti lähes yhtä suuri kuin pidemmän sekvenssin pituus, eli samankaltaisuus on lähellä nollaa. Pisin yhteinen alijonokin laskee kattamaan enää puolet syötteiden pituudesta. Oleellisesti kuitenkin molemmilla tavoilla laskien jo tällaisen, verrattain triviaalin muutoksen jälkeen samankaltaisuus on enää korkeintaan puolet. Mikä tahansa Halsteadin metriikoita käyttävä ominaisuuksien laskentaan perustuva järjestelmä näyttäisi edelleen täyttä vastaavuutta.

Wisen [1993; 1996] esittelemä RKR-GST-algoritmi (*Running Karb-Rabin Greedy String Tiling*) yrittää parantaa vertailun toleranssia transpositioille. Sen sijaan, että etsittäisiin yhtä pitkää yhteistä tekstisegmenttiä, etsitään useita pienempiä. Se pyrkii sovittamaan vertailtavat sekvenssit yhteen limittämällä ne siten, että niistä etsitään mahdollisimman pitkiä ei-päällekkäisiä yhteisiä alijonoja (joissa ei sallita välejä), ja luodaan niiden välille koko algoritmin ajaksi pysyvä assosiaatio (tiling). Ei-päällekkäisyys tarkoittaa sitä, että yksi symboli voi esiintyä korkeintaan yhdessä yhteisessä alijonossa, eli jos se on jo merkattu ”käytetyksi”, se ei voi olla osa toista alijonoassosiaatiota.

Tarkoitus on saada mahdollisimman suuri osa sekvenssien pituudesta katettua mahdollisimman pitkillä yhteisillä alijonoilla. Sekvenssien limitystä havainnollistettu kuvassa 3.



Kuva 3 – Esimerkki yhdestä mahdollisesta sekvenssien limityksestä

Muodostettaessa limitystä, täytyy etsiä yhteisiä alijonoja sekvenssien välillä. Tähän RKR-GST käyttää variaatiota Karpin-Rabinin [1987] -algoritmista, jonka ajatuksena on, että tutkittaessa esiintyykö sekvenssi B sekvenssissä A, voidaan hyväksikäyttää merkkijonojen tiivisteitä (engl. *hash*). Sen sijaan, että jonoa A täytyisi käydä merkki kerrallaan läpi tutkittaessa, alkaako tästä kohtaa mahdollisesti B:n mittainen yhteinen alijono, aina aloittaen uudestaan yhtä positiota pidemmältä kuin edellisellä iteraatiolla, voidaan vertailla vain haettavan merkkijonon tiivisteitä. Eli B:n tiivistettä vertaillaan kaikkien A:n B:n mittaisten alijonojen tiivisteisiin. Tällöinkin alijonojen tiivisteitä joudutaan laskemaan paljon. Siksi Karpin-Rabinin-algoritmi käyttää ns. juoksevaa tiivistefunktiota, joka voidaan muokata helposti edellisen position tiivisteestä uuden laskemisen sijaan (loppuun kuitenkin vain lisätään yksi merkki, ja ensimmäinen putoaa pois). Tiivistefunktion täytyy olla siinä mielessä järkevä, että törmäyksiä ei tapahdu liian usein (”törmäys” tiivisteiden yhteydessä tarkoittaa sitä, että kahdelle eri merkkijonolle muodostuu sama tiiviste). Jos tiivisteet sattuvat olemaan yhtä suuret, algoritmin täytyy nimittäin varmistaa, että merkkijonot ovat oikeasti samat käymällä ne merkki kerrallaan läpi. Algoritmi on hyvä valinta nimenomaan silloin, kun on tarpeen etsiä useampaa mahdollista alijonoa yhtä aikaa ja etsittävän alijonot ovat saman mittaisia, koska kun tietystä A:n positiosta alkavalle alijonolle on kerran muodostettu tiiviste, sitä voidaan vertailla samalla kertaa kaikkiin etsittäviin alijonoihin. Mitä pidempiä alijonoja etsitään, sitä suurempi etu saavutetaan vaihtoehtoisiin merkkijonovertailualgoritmeihin nähden, koska aikavaatimus ei juuri kasva etsittävien alijonojen pituuden mukana.

Wise [1996] hyödyntää näitä Karpin-Rabinin-algoritmin ominaisuuksia etsimällä pidemmästä sekvenssistä ensin kaikkia lyhemmän sekvenssin tietyn mittaisia (alussa pitkiä) alijonoja, ja mikäli niitä löydetään, merkataan niihin kuuluvat positiot kummassakin sekvenssissä ”käytetyiksi”, eli niitä tai niihin ei enää yritetä täsmätä mitään jatkossa. Kun kierros on käyty läpi, vähennetään etsittävien alijonojen pituutta yhdellä ja aloitetaan alusta. Tätä toistetaan kunnes joko kaikki merkit on saatu käytettyä johonkin assosiaatioon tai etsittävien alijonojen pituus laskee alle sellaisen rajan, että sitä lyhemmät vastaavuudet voidaan ohittaa ei-kiinnostavina.

Wisen algoritmi on ahne (greedy), ja kuten sellaisilla on tapana, se ei aina löydä optimaalista limitystä merkkijonojen välillä. Jos lyhyet (vaikka yhden tai kahden symbolin mittaiset) yhteiset sekvenssit karsitaan pois tuloksesta, saattaa algoritmi löytää ja valita tietyltä väliltä yhden pitkän alijonovastaavuuden, vaikka parempi kattavuus saataisiin kahdella lyhyemmällä. Pahimmassa tapauksessa RKR-GST:n aikakompleksisuus on luokkaa $O(n^3)$, mutta Wise toteaa suoritusajan ”todellisen kaltaisella datalla” keskimääräisessä tapauksessa näyttävän lähes lineaariselta.

3.2.3. N-grammit

N-grammi, (usein kutsuttu myös k-grammiksi) on kaikkien sekvenssin n:n peräkkäisen alkion muodostamien alijonojen joukko. Esimerkiksi merkkijonon ”abcdefg” 4-grammi on joukko { ”abcd”, ”bcde”, ”cdef”, ”defg” }, ja 5-grammi joukko { ”abcde”, ”bcdef”, ”cdefg” }. 1-gammia saatetaan kutsua unigrammiksi ja 2-grammia bigrammiksi jne.

N-grammit ovat hyödyllisiä plagioidun koodin tunnistuksessa lähinnä koska, kuten edellä todettiin, symbolisekvenssien samankaltaisuuden arviointi on hankalaa. Jos kuitenkin ajatellaan, että on mahdollista kiinnittää sellainen lukuarvo n, että n kappaletta peräkkäisinä toistuvaa symbolia molemmissa vertailtavissa sekvensseissa on kiinnostava yhtäläisyys, voidaan n-grammeja mielekkäästi käsitellä järjestysriippumattomina joukkoina. Tällöin sekvenssien samankaltaisuuden mittaamiseen on mahdollista soveltaa tilastollisia menetelmiä. Esimerkkinä mainittakoon vaikka Jaccardin indeksi, joka saadaan kaavalla

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} \quad ,$$

missä joukot A ja B ovat vertailtavista lähdekoodeista muodostuneet n-grammit. Tulokseksi saadaan luku yhdestä nollaan (yksi merkitsee täysin samaa joukkoa, ja nolla täysin eri joukkoa). Jos A ja B ovat huomattavan eri kokoiset, Jaccardin indeksiä lienee syytä jollain tavalla suhteuttaa pienemmän joukon kokoon. Muutoin täysin plagioitukaan työ ei tuota suurta arvoa, jos se edustaa vain pientä osaa alkuperäisestä koodista. Vaikka tällainen vertailu ei ole ongelmaton, se on hyvin helppo toteuttaa. Vaihtehtoisia mittareita n-grammien samankaltaisuudelle esittävät esimerkiksi Kondrak [2005] ja (epäsuorasti) Ukkonen [1992].

Kuten Aiken ja muut [2003] korostavat, n:n arvon valinta on kriittistä. Mitä suurempi arvo valitaan, sitä varmempia voidaan olla että n kappaletta peräkkäisiä yhteisiä symboleita ei ole sattumaa, ja väärien positiivisten tulosten määrä pienenee. Liian suuri arvo toisaalta tekee mahdottomaksi n:ää lyhyempien yhtäläisyyksien löytämisen, ja siten altistaa tunnistuksen lyhyiden koodisekvenssien uudelleenjärjestelylle ja uuden koodin lisäykselle. On siis tärkeää valita pienin mahdollinen n, joka kuitenkin suodattaa pois tarpeeksi satunnaisia yhtäläisyyksiä. Yleisesti käytettyjä n:n arvoja tokenisoidulle lähdekoodille tuntuvat olevan esimerkiksi 4 tai 5. Valinta riippuu pitkälti myös tokenisaatiovaiheessa käytetyn kohdeaakkoston koosta. Jos mahdollisia erilaisia symboleita on vähän,

suurempi n on parempi (koska satunnaiset yhtenevät symbolijonot ovat todennäköisempiä). Luonnollista kieltä analysoitaessa pienet arvot, kuten 2 tai 3 (sanaa) tuottavat parempia tuloksia [Barrón-Cedeño & Rosso 2009] – oletettavasti koska aakkosto on useaa kertaluokkaa suurempi.

On huomattava, että n-grammissa on miltei saman verran alkioita kuin alkuperäisessä sekvenssissäkin (n-1 kappaletta vähemmän), ja alkioiden koko on n kertaa suurempi. Tästä johtuen hyvin suurten sekvenssijoukkojen analysointia varten useat kirjoittajat ovat nähneet tarpeelliseksi poimia vain pienen osan koko n-grammin alkioista näytteiksi, eli sormenjäljeksi, joiden perusteella referenssijoukosta löydetään nopeasti samankaltaiset dokumentit. Tästä saadaan uusi ongelma: miten valita näytteet siten, että jos yhdestä sekvenssistä poimitaan jokin n-grammin alkio, vastaava alkio tulee todennäköisesti poimittua myös toisesta sekvenssistä jos se siinä esiintyy?

Yleisesti käytetty tapa valita näytteet vaikuttaa olevan seuraava: muodostetaan jollain tiivistefunktiolla numeerinen tiiviste jokaiselle n-grammin alkiolle, valitaan mielivaltainen (kuitenkin pieni) luku, ja poimitaan dokumenttia edustavaksi sormenjäljeksi kaikki tiivisteet, jotka ovat jaollisia tällä luvulla. Aiken ja muut [2003] kritisoivat tätä lähestymistapaa siitä, että se ei välttämättä löydä joltain, mahdollisesti jopa pitkältä, väliltä yhtään jaollisuusehdon täyttävää alkiota. Tällöin dokumentteihin muodostuu vertailun kannalta kartoittamattomia alueita, joilla esiintyviä samankaltaisuuksia ei voida tunnistaa. He ovat tätä varten kehittäneet algoritmin nimeltä *winnowing* (suom. seulominen).

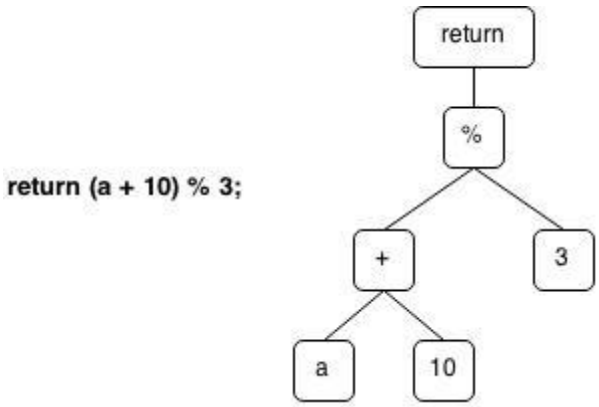
Winnowing vaatii ensin nk. takausarvon kiinnittämisen. Tämä arvo määrittää sen, minkä mittaiselta väliltä algoritmi taatusti valitsee ainakin yhden tiivisteen näytteeksi. N-grammin alkiot jaetaan osittain päällekkäisiin ”ikkunoihin”, joiden koko on $t - n + 1$, missä t on valittu takausarvo, ja n n-grammin alkioiden koko (algoritmin kuvauksessa n:ää on kutsuttu myös ”melukynnykseksi”, jota lyhyempiä yhtäläisyyksiä ei voida eikä haluta huomata). Jokaisesta ikkunasta valitaan näytteeksi pienin siinä esiintyvä tiiviste, ja se lisätään sormenjäljeksi tallennettavien tiivisteiden joukkoon. Tallennettavan alkion arvon lisäksi siitä säilytetään sen esiintymiskohta itse dokumentissa, jotta on mahdollista myöhemmin yhdistää mahdollisesti löytyvät samankaltaisuudet konkreettisiin kohtiin vertailtavissa dokumenteissa. Ikkunaa siirretään yksi (yhden mittainen) askel eteenpäin, ja tästä uudesta ikkunasta valitaan myös pienin arvo. Jos se sattuu olemaan sama kuin aikaisemmin valittu, sitä ei enää lisätä sormenjälkeen. Tämän tarkoitus on vähentää tarvittavien näytteiden määrää poistamalla redundantit alkiot tuloksesta. Pienimmän tiivisteen omaava alkio valitaan siksi, koska se on keskimääräistä todennäköisemmin pienin myös seuraavassa ikkunassa. Kun koko n-grammi on näin käyty läpi, on saatu tulokseksi sormenjälki, joka sisältää ainakin yhden näytteen jokaisesta ikkunasta. Koska valitun alkion tiiviste on aina paikallisesti pienin, voidaan ajatella, että on todennäköistä, että myös toisesta dokumentista tulee valittua samat näytteet. [Aiken *et al.*, 2003]

Mikäli plagiaatintunnistin on asennettu palvelimelle, josta se on avoin suurelle käyttäjäjoukolle, tai referenssijoukko on huomattavan suuri, saattaa näytteiden käyttö olla välttämätöntä. Jos kuitenkin vertailtavien sekvenssien joukko kattaa vain yhden kurssin opiskelijoiden palauttamat

ohjelmointitehtävät, on vaikea kuvitella, että näytteitä käyttämällä saavutettaisiin juuri mitään. Ellei itse vertailualgoritmi ole kohtuuttoman raskas, n-grammit voi ainakin vielä muutaman sadan, noin sata riviä kattavan lähdekoodidokumentin kokoisella joukolla aivan hyvin tallentaa ja käsitellä kokonaan. Tällöin voisi kuvitella saavutettavan sekä saanniltaan että tarkkuudeltaan parempi tunnistus.

3.3. Abstraktit syntaksipuut

Ohjelmakoodi voidaan esittää myös erilaisina puurakenteina. Abstrakti syntaksipuu (*Abstract Syntax Tree,* AST, tai pelkkä syntaksipuu) on usein kääntäjien välikielisenä esitysmuotona käyttämä kuvaus lähdekoodista. AST eroaa konkreettisesta jäsennyspuusta (*Concrete Syntax Tree* tai *Parse Tree*) siten, että se ei eksplisiittisesti sisällä sellaista jäsennyspuun tietoa, joka on muutoinkin nähtävissä puun muodosta (kuten sulkumerkit). Abstrakti syntaksipuu on siis tavallaan redusoitu versio jäsennyspuusta – sen tarkoitus on jättää pois mahdollisimman paljon syntaktisia yksityiskohtia, siten helpottaen lähdekoodin analysointia. Kääntäjä suorittaa tyypillisesti jonkinlaista optimointia lähdekoodille ennen konekielisen ohjelman tuottamista, ja tällainen esitysmuoto on siihen tarkoitukseen sopiva. Sattumalta se on nähty käyttökelpoiseksi myös plagionnin etsimisessä.



Kuva 4 - Lähdekoodirivi ja siitä muodostettu abstrakti syntaksipuu

AST:ssä operandit ovat lehtisolmuissa, operaattorit ja muut kielen avainsanat sisäsolmuissa. Kuvassa 4 on esitetty ohjelmakoodi ja siitä tuotettu syntaksipuu. Puun muodostaminen ei ole yksinkertainen tehtävä: lähdekoodi pitää ensin skannata, tokenisoida, jäsentää (oikein) yms. Onneksi kääntäjät tekevät nämä operaatiot joka tapauksessa, ja joskus tarjoavat keinoja tuottamansa AST:n tarkasteluun. Esimerkiksi tuoreemmat versiot Java-kielestä sisältävät rajapinnan, jonka kautta on mahdollista ohjelmallisesti kutsua kääntäjämoduulia mielivaltaiselle lähdekoodille, ja sen jälkeen tarkastella siitä muodostettuja konstruktioita – kuten juuri abstraktia syntaksipuuta. Tällä tavoin hankittuja puita täytyy tosin hieman karsia ennen kuin niitä voi mielekkäästi vertailla keskenään.

Kuvan 4 koodirivistä saadaan nimittäin Javan kääntäjältä kysymällä lähes sata solmua käsittävä syntaksipuu. Suurin osa siitä on tässä tapauksessa täysin turhaa tietoa, jonka sisällyttäminen vertailtaviin puihin vain tarpeettomasti monimutkaistaisi vertailuvaihetta.

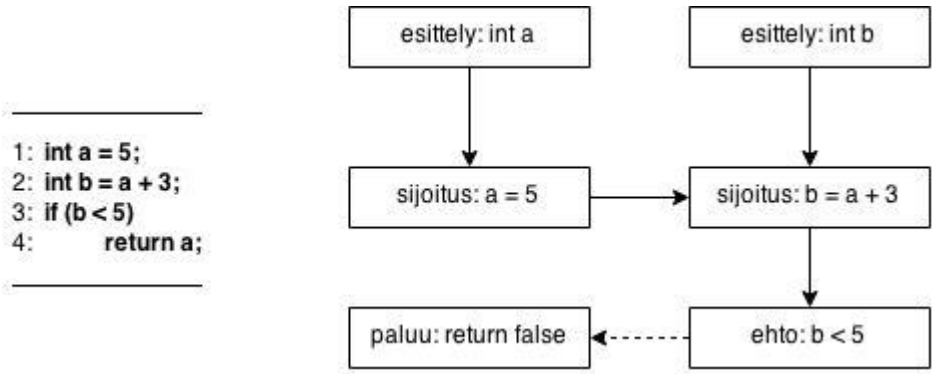
Lähdekoodiparista muodostettujen puiden samankaltaisuuden mittarina voidaan käyttää esimerkiksi niiden välistä muutosetäisyyttä (*tree edit distance*). Samaan tapaan kuin Levenshteinin etäisyys merkkijonojen tapauksessa, puiden välinen muutosetäisyys tarkoittaa pienintä mahdollista määrää tietystä joukosta valittuja operaatioita, joilla vertailtavista puista saadaan keskenään yhtenevät. Sallittuja operaatioita ovat solmun lisäys, poisto ja korvaus. Jos solmu poistetaan, sen lapset siirretään sen vanhemman lapsiksi. Jos solmu lisätään, kaikki sen oikealla puolelle jäävät sisarsolmut siirretään sen lapsiksi. Solmun korvauksessa vain vaihdetaan solmun nimi, mutta puun rakennetta ei muutoin muuteta. Tarvittaessa eri operaatioille voidaan määrittää painoarvo, tai ”hinta”, jolloin koko muutosoperaatiosekvenssin pituus on siinä esiintyvien yksittäisen operaatioiden painoarvojen summa. Puiden välisen muutosetäisyyden laskemiseen on olemassa useita tapoja. Tarkoitukseen voidaan käyttää esimerkiksi Pawlikin ja Augstenin [2011] menetelmää, tai Zhangin ja Shashan [1989] tai Demainen ja muiden [2009] esittelemiä algoritmeja.

Harmillisesti syntaksipuu ei pysy alkuunkaan muuttumattoma, mikäli koodiin sovelletaan joitain jo aikaisemmin kuvattuja naamiointimenetelmiä. Jos jonkin kontrollilauseen, vaikkapa *if*- tai *for*-rakenteen runko sisältää vain yhden lausekkeen, useassa ohjelmointikielessä se voidaan sulkea kaarisulkeiden sisälle tai olla sulkematta. Jo vaihtelu tässä vaikuttaa puun rakenteeseen sen verran, että rungolle muodostuu tai on muodostumatta oma solmunsa. Iteraatiorakenteen korvaaminen toisella tai turhan koodin lisääminen vääristää puuta vieläkin enemmän. Ligaarden [2007] kuvaa yksityiskohtaisesti eri naamiointimenetelmien vaikutusta syntaksipuihin.

Ligaardenin oma lähestymistapa on löytää sellainen vertailualgoritmi, jolla on sopiva määrä toleranssia vaihteluille puun muodossa. Aihetta ovat tutkineet myös Tao ja muut [2013], jotka pyrkivät ratkaisemaan ongelman esiprosessoimalla lähdekoodia kehittämiensä heuristiikkojen pohjalta (esimerkiksi ehtolauseiden operaattorit vaihdetaan aina ensin vakiojärjestykseen), jotta semanttisesti yhtenevien lähdekoodien puut olisivat vertailuvaiheessa mahdollisimman samankaltaiset.

3.4. Riippuvuusgraafit

Riippuvuusgraafi (Program Dependence Graph, PDG) [Ottenstein & Ottenstein, 1984] on tapa esittää ohjelman proseduurin sisäisiä riippuvuussuhteita joukkona erityyppisiä solmuja ja suunnattuja kaaria. Graafin solmut kuvaavat koodissa esiintyviä lausekkeita, kuten muuttujan esittelyjä, sijoituksia tai ehtolauseita. Kaaret symboloivat joko data- tai kontrolliriippuvuutta kahden solmun välillä. Datariippuvuus tarkoittaa, että operaation oikea tulos riippuu toisen operaation suorittamisesta ensin. Kontrolliriippuvuudella puolestaan tarkoitetaan sitä, että koko solmussa olevan operaation suoritus tai suorittamatta jättäminen riippuu suoraan toisen lausekkeen arvioinnin tuloksesta.

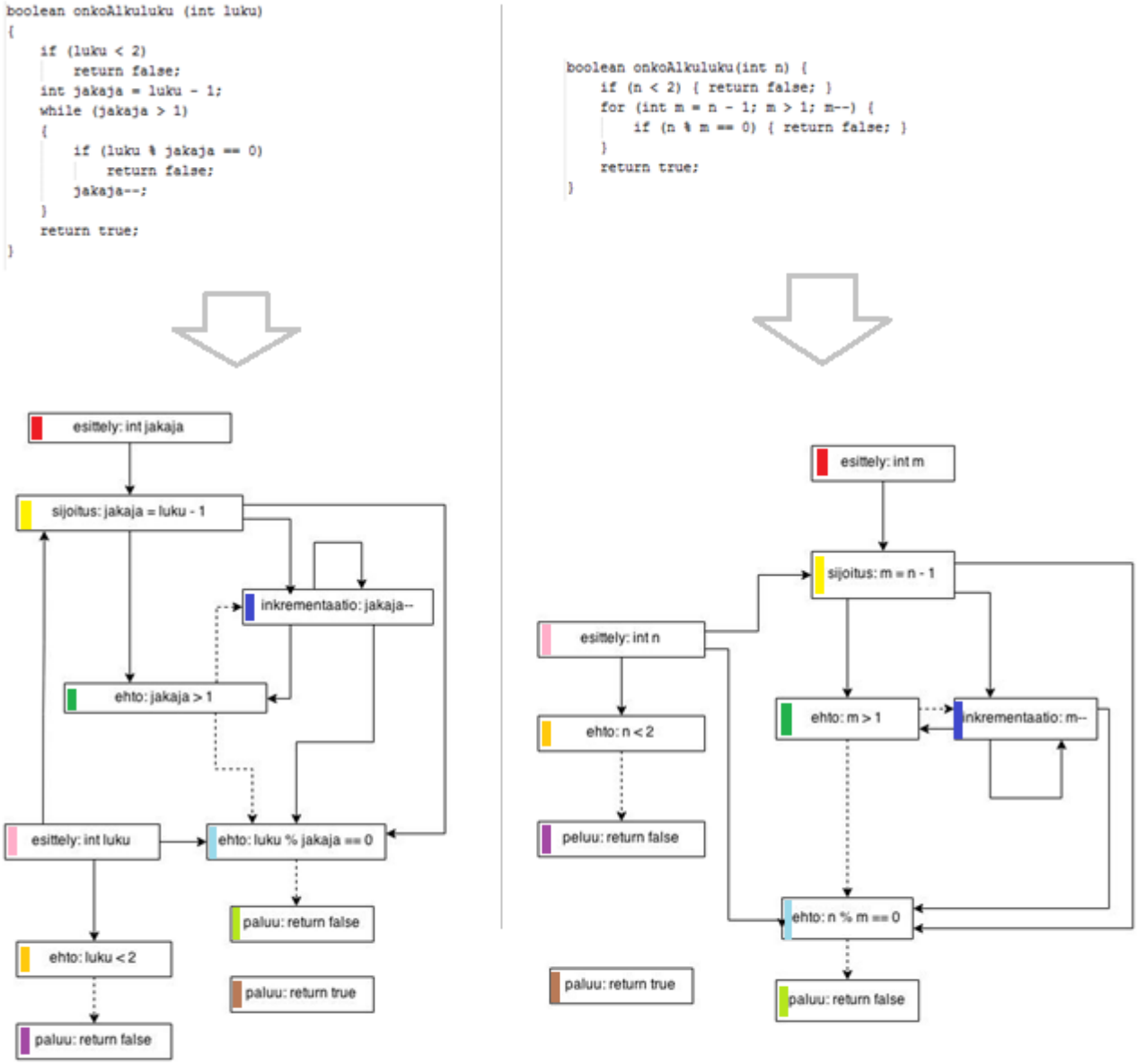


Kuva 5 - Lähdekoodirivejä esitetty sekä sellaisenaan että riippuvuusgraafina

Kuvassa 5 on esimerkinomaisesti neljästä lähdekoodirivistä muodostettu riippuvuusgraafi. Siinä on käytetty samaa notaatiota kuin esimerkiksi Liun ja muiden [2006] tekstissä: katkoviiva kuvaa solmujen välistä kontrolliriippuvuutta ja jatkuva viiva datariippuvuutta. Solmun tyyppi ja sisältö on erotettu toisistaan kaksoispisteellä. Rivin 2 sijoituslauseella on datariippuvuus rivin 1 operaatiosta (muuttujaan *b* saatava arvo riippuu edellisellä rivillä asetetusta muuttujan *a* arvosta). Rivillä 4 on kontrolliriippuvuus rivistä 3: rivi 4 suoritetaan vain, mikäli rivin 3 ehtolause on totta.

Vaikka ensimmäisen konkreettisen plagiaarintunnistimen esitellyt K. Ottenstein on ollut mukana myös kirjoittamassa varhaisimpia riippuvuusgraafeja käsitteleviä tekstejä [Ottenstein & Ottenstein, 1984; Ferrante *et al.,* 1987], ei niissä mainita mitään plagiaatintunnistuksesta. Ehkä tämä johtuu tuolloisten tietokoneiden laskentakapasiteetin rajoituksista. Prosessoriaika oli kallista, ja monet 1970-luvun lopun ja 1980-luvun alun tutkijoista pitivät tärkeänä mainita, kuinka monta dollarisenttiä yhden lähdekoodiriviparin käsittely suunnilleen maksaa milläkin menetelmällä. 2000-luvulle tultaessa kuitenkin useampikin kirjoittaja esittää, että ohjelman sisäisiä riippuvuuksia kuvaavaa PDG voisi soveltua myös lähdekoodisekvenssien samankaltaisuuden arviointiin. [Krinke *et al.,* 2001; Liu *et al.,* 2006]

Liu ja muut [2006] tutkivat tätä tarkastelemalla aligraafien isomofrismia. Graafien G ja H sanotaan olevan keskenään isomorfisia, jos ne ovat rakenteeltaan identtiset, kun ei huomioida solmujen sisältämää informaatiota. Tarkempi määritelmä isomorfismille on, että on graafien solmujoukkojen välillä on (ainakin yksi) bijektio, funktio *f,* jonka avulla graafin G solmut voidaan asettaa yksi-yhteen vastaamaan graafin H solmuja siten, että jos kaksi solmua on vierekkäiset graafissa G, niin niiden vastinsolmut ovat vierekkäiset graafissa H. Graafi G on aligraafi-isomorfinen graafin H kanssa, mikäli jokin G:n aligraafi on isomorfinen H:n kanssa. Ajatuksena on, että jos ohjelma on plagiaatti toisesta, siitä muodostettu riippuvuusgraafi on aligraafi-isomorfinen alkuperäisen ohjelman kanssa (ks. kuva 6).



Kuva 6 – Lähdekoodista ja sen plagioidusta versiosta muodostetut riippuvuusgraafit. Solmujen värit kuvaavat graafien väliltä löytyvän bijektion vastinsolmuja.

Koska PDG:t ovat merkittyjä ja suunnattuja graafeja, täytyy bijektiota etsittäessä ottaa huomioon muutamia lisäseikkoja. Vastinsolmujen täytyy olla samaa tyyppiä, ja niiden ja niiden viereisten solmujen välisten kaarten pitää ensinnäkin kulkea oikeaan suuntaan, ja olla myös oikeaa tyyppiä vastinkaartensa kanssa. Datariippuvuuden vastinkaari ei siis voi olla tyyppiä kontrolliriippuvuus, eikä ”sijoitus”-tyyppistä solmua voi toisessa graafissa vastata ”paluu”-tyyppinen solmu, tai vertailun tarkkuus heikkenee.

PDG on sikäli sopiva esitysmuoto plagiaatintunnistuksen kannalta, että lausekkeiden välisiä riippuvuussuhteita on hankala muokata ilman syvempää ymmärrystä alkuperäisestä lähdekoodista, ja ne sellaisenaan kuvaavat hyvin ohjelman ”ytimen”. Toisistaan riippumattomien koodisekvenssien uudelleenjärjestely tai tunnisteiden uudelleennimeäminen ei vaikuta ohjelmasta syntyvään riippuvuusgraafiin mitenkään. Toiminnallisesti turhien kohtien lisääminen kyllä laajentaa graafia, mutta sen kopioitua osaa vastaava aligraafi säilyy silti isomorfisena alkuperäisen ohjelman PDG:n kanssa. Muuttujien uudelleenkäyttö sen sijaan muuttaa graafiesitystä. Jos esimerkiksi jo käyttötarkoituksensa toteuttanut ja turhaksi käynyt muuttuja kierrätetään saman skoopin sisällä uuteen tarkoitukseen, näkyy se graafissa uutena datariippuvuutena. Täydellistä aligraafi-isomorfismia ei siis voi vaatia, vaan vertailun on sallittava jonkinlainen vaihtelu. [Liu *et al.,* 2006]

Graafien isomorfismin etsiminen on valitettavasti NP-täydellinen ongelma – ei ole olemassa tunnettua algoritmia, jolla se voitaisiin taatusti ratkaista polynomiaalisessa ajassa. Krinke ja muut [2001] raportoivat kirjoittamansa ohjelman käyttäneen 38848 sekuntia (noin 10 tuntia) etsiessään duplikoituja osia yhden ainoan, 3968 koodiriviä käsittäneen ohjelman sisältä.

Liun ja muiden [2006] ratkaisu on yrittää karsia vertailujoukosta sellaiset graafit, jotka eivät ansaitse lähempää tarkastelua. He kuvaavat kaksi eri tapaa suodattaa tuloksia. Toinen on ”häviötön” siinä mielessä, että se ei ohita yhtäkään potentiaalisesti kiinnostavaa graafiparin vertailua, vaan ainoastaan esimerkiksi sellaiset, joissa toinen graafi on suurempi kuin siitä mahdollisesti plagioidun dokumentin graafi (tällöin jälkimmäinen ei voi mitenkään olla aligraafi-isomorfinen edellisen kanssa). Toinen menetelmä, jota he kutsuvat ”häviölliseksi suodattimeksi”, suorittaa jonkinlaista etukäteisarviointia siitä, ovatko graafit mahdollisesti tutkimisen arvoiset. Se saattaa olla väärässä (mistä siis nimi ”häviöllinen”), ja jättää joitakin oikeasti aligraafi-isomorfisia pareja tutkimatta. Tutkijaryhmä raportoi hyvin vaikuttavia aikasäästöjä, kun molempia suodattimia käytettiin yhdessä. Samoiksi tunnistetuista tuloksista hukattiin prosessissa noin 9 prosenttia, mikä ei kuulosta paljolta.

3.5. Latentti semanttinen analyysi

Latentti semanttinen analyysi (engl. *Latent Semantic Analysis*, LSA) [Dumais *et al.,* 1988] on pääsääntöisesti luonnollisen kielen analysoinnissa käytetty menetelmä. Sen suunnittelun lähtökohtana on ollut löytää ratkaisu ongelmaan, joka syntyy siitä, että usein kirjoittajat käyttävät täysin eri sanoja saman asian ilmaisemiseen (synonyymit), ja täysin samat sanat taas saattavat tarkoittaa useaa eri asiaa (polysemia, homonyymit). Lauseita on tietenkin ohjelmallisesti vaikea tunnistaa samoiksi, jos niissä käytetyt sanat ovat erilaisia, vaikka semantiikka niiden taustalla olisikin yhtenevä.

LSA perustuu aikaisempaan tiedonhaun tekniikkaan nimeltä *Vector Space Model*, jossa hakuavaruus ja siihen kohdistuvat kyselyt esitetään vektoreina. Vertailtavista dokumenteista muodostetaan matriisi, jonka rivit vastaavat kaikkia uniikkeja dokumenteissa esiintyviä sanoja (tai ”termejä”), ja sarakkeet itse dokumentteja. Soluihin kirjataan tieto siitä, kuinka monta kertaa rivin

termi esiintyy sarakkeen dokumentissa. Kun tästä matriisista halutaan löytää joidenkin termien perusteella samankaltaisia dokumentteja, hakutermeistä voidaan muodostaa vektori, ja matriisin dokumenttivektoreita (eli sarakkeita) vertailla hakuvektoriin niiden välisen kulman kosinia mittarina käyttäen. Se lähestyy yhtä, mitä lähempänä dokumenteissa esiintyvien termien suhteelliset määrät ovat toisiaan. [Cosma & Joy, 2012]

LSA laajentaa tätä konseptia siten, että se yhdistää sellaiset termit (matriisin rivit), joiden se arvelee olevan semanttisesti samankaltaisia. Yhdistämiseen käytetään pääakselihajotelma-nimistä menetelmää, johon en tässä puutu sen tarkemmin. Semanttinen samankaltaisuus päätellään termien kontekstista: jos dokumenteissa jokin termipari esiintyy usein samojen termien läheisyydessä, tehdään oletus, että kyseessä on merkitykseltään samankaltaiset termit. Esimerkiksi jos käsiteltävässä aineistossa termiä ”Tampereen” usein seuraa termi ”yliopisto” tai ”teatteri”, voimme päätellä, että ”yliopisto” ja ”teatteri” ovat synonyymejä. Tai ainakin merkitykseltään melko lähellä toisiaan.

Cosma & Joy [2012] ovat yrittäneet soveltaa tällaista samankaltaisuuden arviointimenetelmää myös lähdekoodille. Heidän ratkaisunsa, erona luonnolliseen kieleen, esiprosessoi lähdekoodista pois kommentit ja joitain Javan syntaksiin kuuluvia termejä. Mielenkiintoisesti kuitenkin esimerkiksi tunnisteiden nimet jätetään rauhaan. Tästä voisi olettaa aiheutuvan sen, että saman niminen tunniste käytettynä eri yhteydessä eri lähdekoodidokumentissa sekoittaisi semanttisten suhteiden luomista.

Samoin kuin riippuvuusgraafien tapauksessa, yhtään LSA:ta hyödyntävää lähdekoodille suunniteltua plagiaatintunnistustyökalua ei löytynyt. Menetelmän pätevyyttä voi siis vain arvailla kirjallisuuden perusteella. Intuitiivisesti voisi ajatella, että se ei välttämättä suoriudu kovin hyvin. Ensinnäkin, kysehän oikeastaan on monimutkaistetusta ominaisuuksien laskennasta. Laskettavien ominaisuuksien joukko vain ei ole vakio, se on sama kuin dokumenteissa esiintyvien termien joukko. Lisäksi tulokseen vaikuttavat enemmän ominaisuuksien suhteellinen kuin absoluuttinen määrä (koska vektorien väliseen kulmaan ei vaikuta vektorien pituus vaan suunta). Toiseksi, ohjelmointikielissä ei pääsääntöisesti juurikaan esiinny synonyymejä. Monet termit ovat kyllä semanttisesti lähellä toisiaan, (kuten *while* ja *for*), mutta on kyseenalaista, voiko tämän menestyksekkäästi päätellä niiden konteksteista. Ehkäpä tällaisia termien välisiä assosiaatioita voisi luoda etukäteen käsin, kun ne kuitenkin tiedetään, ja niitä on melko rajallinen joukko?

3.6. Tulosten esittäminen käyttäjälle

Kun varsinainen vertailu on tavalla tai toisella suoritettu, tarvitsee sen tulokset vielä kommunikoida ohjelman käyttäjälle. Tämän pitäisi niiden perusteella pystyä, mielellään nopeasti, muodostamaan kattava kuva siitä, mitkä dokumentit ansaitsevat lähempää tarkastelua ja miksi ne on merkattu epäilyttäviksi.

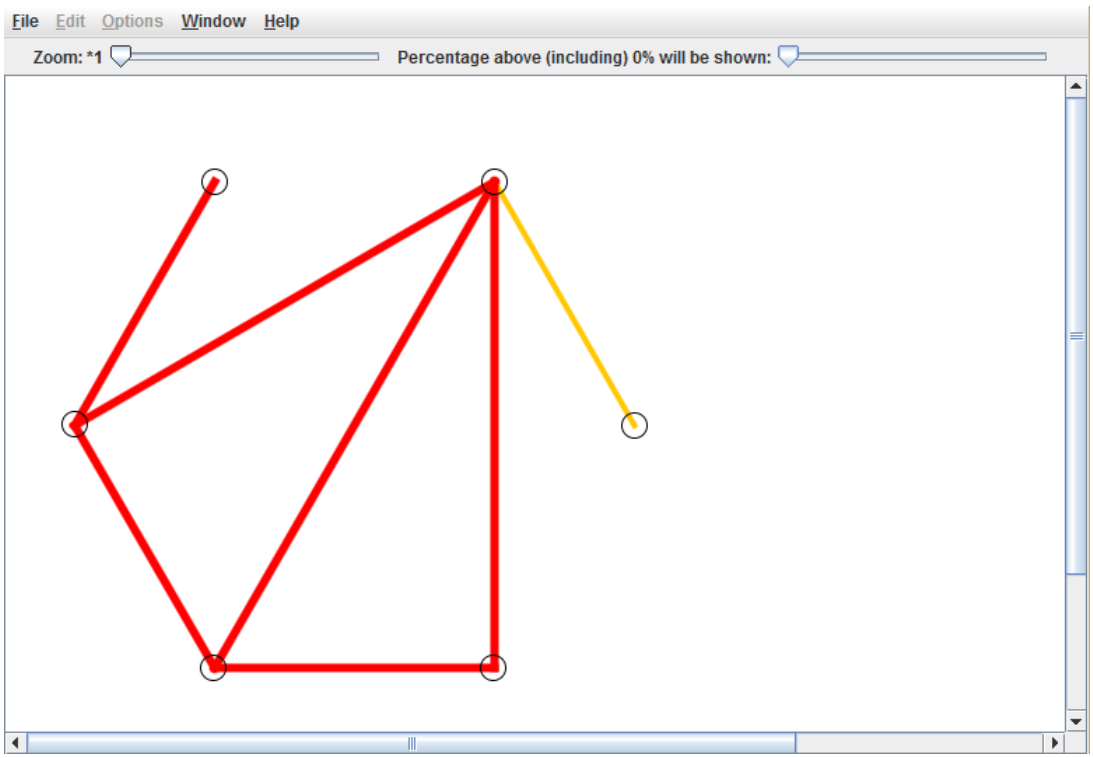
Tyypillisesti vertailun tuloksesta muodostetaan samankaltaisuuden mukaan järjestetty lista ohjelmakoodipareista, josta ehkä poistetaan jonkin määritellyn kynnysarvon alittavat, ei niin kiinnostavat parit. Näin huipulle jäävät todennäköisimmät, tarkempaa huomiota ansaitsevat

plagiaattikandidaatit. Jotta käyttäjän on mahdollista nähdä mistä samankaltaisuus on syntynyt, useimmat kohdassa 4.1 esitellyistä työkaluista tarjoavat jonkinlaisen mahdollisuuden valita listalta tietty alkiopari yksityiskohtaisempaan tarkasteluun, kuten kuvassa 7.

Kuva 7 –Tiedostopari yksityiskohtaisemmassa tarkastelussa. Täsmääviksi tunnistetut koodisegmentit on värjätty punaiseksi (MOSS)

Tämä ei kuitenkaan välttämättä parhaimmalla tavalla havainnollista koko kuvaa tunnistuksesta. On esimerkiksi mahdollista, että lähdekoodit jakavat saman alkuperän, mutta eivät ole keskenään kiinnostavan samanlaisia – eli vaikkapa ohjelmat B ja C on molemmat plagioitu ohjelmasta A, mutta B:n ja C:n keskinäinen samankaltaisuus on vähäistä. Jotta tällaiset piirteet saadaan ilmaistua käyttäjälle tehokkaasti, voitaisiin tuloksesta listausnäkymän lisäksi muodostaa esimerkiksi graafi, jossa vertailussa mukana olleet lähdekoodit ovat solmuina, ja niiden välisten kaarien pituus kääntäen verrannollinen niiden samankaltaisuuteen. Selkeyden vuoksi kannattanee karsia pois kaaret, joiden pituus on liian suuri ollakseen kiinnostava.

Tätä ajatusta jossain määrin toteuttaa esimerkiksi Sherlock, joka tarjoaa tuloksistaan visuaalisen esityksen (kuvassa 8). Se ei kuitenkaan graafia piirtäessään ota huomioon kaarien pituuksia, vaan kaikki vertailussa mukana olleet ohjelmat ovat solmuina saman ympyrän kehällä, ja samankaltaisuuden vahvuus ilmaistaan vain kaarien värillä, vieläpä aika suppealla skaalalla (kaari voi olla joko punainen tai keltainen).



Kuva 8 - Samankaltaiset lähdekooditiedostot esitettynä graafina (Sherlock)

4. Nykyisiä työkaluja ja niiden testausta

Tässä luvussa esittelen lyhyesti muutamia lähdekoodiplagiarismin tunnistuksessa käytettyjä konkreettisia sovelluksia sekä niiden laskemia samankaltaisuustuloksia Java-kielisille lähdekooditiedostoille, joihin on sovellettu eriasteista plagioinnin naamiointia. Kaikki käsitellyt ohjelmat käyttävät referenssijoukkona yksinomaan käyttäjän valitsemia tiedostoja, eli yksikään niistä ei esimerkiksi etsi yhtäläisyyksiä internetistä tai omista tietokannoistaan. Monet niistä sisältävät mahdollisuuden käyttää ”perustapauksia”, eli määritellä tiedostoja, joiden sisältämän koodin käyttö on hyväksyttävää, eikä sen esiintymistä dokumenteissa siten lueta epäilyttäväksi.

4.1. Työkaluja plagioinnin tunnistukseen

Measure Of Software Similarity (MOSS) on Stanfordin yliopistossa kehitetty ja laajalti alan kirjallisuudessa käsitelty lähdekoodien samankaltaisuutta arvioiva ohjelma. MOSS perustuu n-grammien alkioista (kohdassa 3.2.2 käsitellyllä) winnowing-algoritmilla poimittujen näytteiden vertailuun. Ohjelman lähdekoodi on suljettua, mutta Aiken ja muut [2003] kuvaavat sen toimintaa periaatteellisella tasolla. Paikallinen asennuspaketti MOSS:sta on tekijöiden mukaan saatavilla pyynnöstä, mutta normaalisti ohjelma toimii palvelutyyppisesti lähettämällä vertailtavat ohjelmakoodit ja halutut parametrit palvelimelle, jossa varsinainen vertailu suoritetaan. Vastausviestinä käyttäjä saa internet-osoitteen, josta tulokset voi käydä katsomassa. Palvelu vaatii ilmaisen rekisteröinnin ja jonkinlaisen lähetyskäyttöliittymän pystyttämisen (näitä on saatavilla useita). MOSS:in tukemien ohjelmointikielten joukko on huomattava: C, C++, Java, C#, Python, Visual Basic, Javascript, FORTRAN, ML, Haskell, Lisp, Scheme, Pascal, Modula2, Ada, Perl, TCL, Matlab, VHDL, Verilog, Spice, MIPS assembly, a8086 assembly, a8086 assembly, MIPS assembly ja HCL2. MOSS löytyy osoitteesta https://theory.stanford.edu/~aiken/moss/.

Yet Another Plague (YAP) käyttää lähdekoodin analysointiin tokenisaatiopohjaista lähestymistapaa. YAP:ista on kolme eri versiota (1, 2 ja 3). Tuorein näistä, YAP3, toteuttaa kohdassa 3.2.3 käsitellyn RKR-GST-algoritmin. Valitettavasti YAP ei sisällä Java-kielistä tokenisaattoria, joten sitä ei voitu järkevästi sisällyttää samaan testiin muiden kanssa (tuetut kielet ovat C, PASCAL ja LISP). Ohjelman lähdekoodi on saatavilla. Vaikka YAP ei ole mukana varsinaisessa vertailussa, voitaisiin olettaa sen menestyvän suunnilleen yhtä hyvin kuin JPlag, koska käytetyt menetelmät ovat hyvin pitkälti samoja. YAP:n voi ladata osoitteesta http://luggage.bcs.uwa.edu.au/~michaelw/YAP.html.

JPlag sai alkunsa Karlsruhen yliopistossa opiskelijaprojektina vuonna 1996. Toimintaperiaatteiltaan se on hyvin samanlainen kuin YAP3. JPlag vertailee lähdekoodeista tokenisoituja symbolisekvenssejä käyttäen variaatiota Wisen [1993] RKR-GST algoritmista. Tarkemmin työkalua ja sen toimintaa kuvaavat Malpohl ja muut [2002]. JPlag tukee Java, C#, C, C++ ja Scheme –kielisten dokumenttien vertailua. Aikaisemmin ohjelma toimi internet-palveluna, johon täytyi rekisteröityä, mutta nykyinen versio siitä asennetaan paikallisesti. JPlagin lähdekoodi on saatavilla, mutta ei avointa. JPlag löytyy osoitteesta https://jplag.ipd.kit.edu/.

The software and text similarity tester (SIM) on hieman harvemmin kirjallisuudessa mainittu työkalu. Tekijöiden, Grunen ja Huntjensin [1989] kuvauksen perusteella käytetty vertailumenetelmä vaikuttaa samankaltaiselta kuin mm. YAP3:ssa tai Jplagissa. Ohjelmakoodeista etsitään yhteisiä alijonoja, jotka saattavat olla limittäin, mutta eivät päällekäin (aivan kuten RKR-GST:ssä). Ohjelman lähdekoodi on saatavilla, mutta ei avointa. SIM tukee C,- Java-, Pascal-, Modula-2-, Miranda-, ja Lisp–kielisten ohjelmakoodidokumenttien vertailua. Käyttöliittymä on komentorivipohjainen, mikä ei ole yllättävää ottaen huomioon että SIM on tämän tutkielman kirjoitushetkellä noin 26 vuotta vanha. Se on saatavilla osoitteesta http://dickgrune.com/Programs/similarity_tester/.

Sherlock (1) on Rob Piken kirjoittama yksinkertainen työkalu tiedostojen samankaltaisuuden vertailuun. Ohjelman lähdekoodi on saatavilla, ja sen tarkastelun perusteella toimintaperiaate näyttää olevan seuraavanlainen: vertailtavista dokumenteista poimitaan muutaman peräkkäisen ”sanan” mittaisia kohtia, joista muodostetaan tiivisteet. Tiivisteistä osa heitetään pois tietyn logiikan mukaan, ja jäljelle jääneitä kutsutaan tiedoston sormenjäljeksi. Tiedostojen samankaltaisuuslukema muodostetaan sen mukaan, paljonko niiden sormenjäljet sisältävät yhteisiä tiivisteitä verrattuna niiden käsittämien tiivisteiden kokonaismäärään. Vertailu on puhtaasti tekstipohjainen, mutta tekijän mukaan sitä on mielekästä käyttää myös lähdekoodille. Sherlockin voi ladata osoitteesta http://sydney.edu.au/engineering/it/~scilect/sherlock/.

Sherlock (2) syntyi Warwickin yliopiston tutkimustyöstä vuonna 1998. Ilmeisesti tällä Sherlockilla ei ole nimeään lukuunottamatta mitään yhteistä edellä esiteltyyn työkaluun. Ohjelma vertailee sille syötettyjä lähdekoodeja kolmessa muodossa: alkuperäisessä, kommenteista ja turhista välilyönneistä riisutussa, sekä tokenisoidussa. Tarkemmin sen toimintaa kuvaavat Joy ja Luck [1999]. Sherlock (2) lähdekoodeinen on saatavissa osoitteesta http://www2.warwick.ac.uk/fac/sci/dcs/research/ias/software/sherlock/.

Plaggie on Helsingin Yliopistossa kehitetty, vuonna 2006 julkaistu avoimen lähdekoodin järjestelmä. Se on suunniteltu analysoimaan nimenomaan opiskelijoiden palauttamia Java-kielisiä ohjelmia. Ahtikainen ja muut [2006] kuvaavat ohjelman toimintaperiaatteita. Vaikka Plaggien lähdekoodi on saatavilla, en onnistunut kääntämään sitä, mistä syystä se on valitettavasti jäänyt pois vertailusta. Sama ongelma Plaggien kanssa on ollut esimerkiksi Martinsilla ja muilla [2014]. Plaggie löytyy osoitteesta https://www.cs.hut.fi/Software/Plaggie/.

4.2. Työkalujen suorituskyvyn arviointia

Seuraavissa kappaleissa tutkitaan pienimuotoisen kokeen avulla, kuinka hyvin edellä kuvatut järjestelmät itseasiassa tunnistavat plagiarismia (tai ehkä paremminkin: minkäasteista lähdekoodin kopioinnin peittelyä on tehtävä, jotta ne eivät sitä tunnistaisi). Koska olin jo kirjoittanut yksinkertaisen Java-tokenisaattorin kuvan 2 muodostamista varten, lisäsin siihen pisimpiä yhteisiä alijonoja laskevan vertailutoiminnon, annoin sille nimen (Tremendously Effective Similarity Tool, TEST) ja sisällytin tämänkin ohjelman mukaan kokeeseen.

4.2.1. Käytetty data

Kokeessa on käytetty kahdeksaa eri Java-kielistä lähdekooditiedostoa, jotka on merkitty kirjaimilla A-H. Näyte A (kuvassa 9) on lähtökohta, johon on sovellettu Whalen [1990] listaamia naamiointimenetelmiä kasvavassa järjestyksessä näytteissä B-F (kuvat 10-14). Muutokset eivät ole kumulatiivisia, eli esimerkiksi C ei sisällä B:n muutoksia A:han nähden. Poikkeus tähän on lähdekoodi F, jossa on mukana kaikki edellisten muutokset. Lähdekoodit G ja H (kuva 15) ovat lähinnä kontrollinäytteitä, eikä niillä ole mitään varsinaista yhteyttä A:han, lukuunottamatta suunnilleen samaa rivimäärää.

```
public class A {
    private static char[][] field;
    private static char pen = '*';
    private static int tPosX = 1;
    private static int tPosY = 1;
    private static String tDir = "right";
    public static void showField() {
        int rivi = 0;
        int sarake = 0;
        while (rivi < field[0].length) {
            sarake = 0;
            while (sarake < field.length) {
                System.out.print(field[sarake][rivi]);
                sarake = sarake + 1;
            }
            System.out.println();
            rivi = rivi + 1;
        }
        System.out.println("Turtle is located at "
                + tPosX + "," + tPosY
                + ", moving " + tDir);
    }
    public static void moveTurtle(int steps) {
        while (steps > 0) {
            field[tPosX][tPosY] = pen;
            if (tDir.equals("up"))
                tPosY = tPosY - 1;
            else if (tDir.equals("right"))
                tPosX = tPosX + 1;
            else if (tDir.equals("down"))
                tPosY = tPosY + 1;
            else
                tPosX = tPosX - 1;
            steps = steps - 1;
        }
        field[tPosX][tPosY] = 'x';
    }
}
```

A

Kuva 9 - Lähdekoodi A. Ratkaisu perustason ohjelmointikurssin viikkoharjoitustehtävään.

```
public class B
{
    private static char[][] kentta;
    private static char kyna = '*';
    private static int pX = 1;
    private static int pY = 1;
    private static String suunta = "right";
    public static void naytaKentta()
    {
        int i = 0;
        int j = 0;
        while (i < kentta[0].length)
        {
            j = 0;
            while (j < kentta.length)
            {
                System.out.print(kentta[j][i]);
                j = j + 1;
            }
            System.out.println();
            i = i + 1;
        }
        System.out.println("Turtle is located at "
                + pX + "," + pY
                + ", moving " + suunta);
    }
    public static void liikutaKonnaa(int askelia)
    {
        while (askelia > 0)
        {
            kentta[pX][pY] = kyna;
            if (suunta.equals("up"))
                pY = pY - 1;
            else if (suunta.equals("right"))
                pX = pX + 1;
            else if (suunta.equals("down"))
                pY = pY + 1;
            else
                pX = pX - 1;
            askelia = askelia - 1;
        }
        kentta[pX][pY] = 'x';
    }
}
```

B

Kuva 10 - Lähdekoodi B. Leksikaalisia muutoksia.

Kuvassa 10 esitetty näyte B on esimerkki hyvin yksinkertaisesta leksikaalisesta plagioinnin peittelystä. Joitain rivinvaihtoja on lisätty, ja kaikki tunnisteet on uudelleennimetty (kaksi ensimmäistä kohtaa Whalen [1990] listalta). Voisi ajatella, että tällaisilla muutoksilla ei vielä hämätä yhtäkään plagioinnintunnistustyökalua.

```
public class C {
    private static char[][] field;
    private static char pen = '*';
    private static int tPosX = 1;
    private static int tPosY = 1;
    private static String tDir = "right";
    public static void showField() {
        int rivi = 0;
        int sarake = 0;
        while (field[0].length >= rivi) {
            sarake = 0;
            while (field.length >= sarake) {
                System.out.print(field[sarake][rivi]);
                sarake = 1+sarake;
            }
            System.out.println();
            rivi = 1+rivi;
        }
        System.out.println("Turtle is located at "
                + tPosX + "," + tPosY
                + ", moving " + tDir);
    }
    public static void moveTurtle(int steps) {
        while (0 <= steps) {
            field[tPosX][tPosY] = pen;
            if ("up".equals(tDir))
                tPosY = tPosY-1;
            else if ("right".equals(tDir))
                tPosX = 1+tPosX;
            else if ("down".equals(tDir))
                tPosY = tPosY+1;
            else
                tPosX = tPosX-1;
            steps = steps-1;
        }
        field[tPosX][tPosY] = 'x';
    }
}
```

C

Kuva 11 - Lähdekoodi C, operandit käännetty

Myöskään näyte C (kuva 11) ei sisällä kovin perustavanlaatuisia muutoksia. Vaihdettavissa olevat operandit on siirretty toiselle puolen operaattoreita. Esimerkiksi *rivi=rivi+1;* on muuttunut muotoon *rivi=1+rivi;* ja *tDir.equals(”right”)* on nyt *”right”.equals(tDir).* ”Pienempi kuin” -vertailuoperaattorit on muutettu ”suurempi tai yhtäsuuri kuin” -operaattoreiksi samalla kun niiden oikea ja vasen puoli on vaihdettu keskenään.

```
public class D {
    private static String tDir = "right";
    private static int tPosY = 1;
    private static char pen = '*';
    private static int tPosX = 1;
    private static char[][] field;
    private static boolean debug = false;
    public static void moveTurtle(int steps) {
        if (debug)
            System.out.println("@moveturtle");
        while (steps > 0) {
            field[tPosX][tPosY] = pen;
            if (tDir.equals("right"))
                tPosX = tPosX + 1;
            else if (tDir.equals("down"))
                tPosY = tPosY + 1;
            else if (tDir.equals("up"))
                tPosY = tPosY - 1;
            else if (tDir.equals("zap"))
                tPosY = tPosY - 1;
            else
                tPosX = tPosX - 1;
            tPosX = tPosX * 1;
            steps = steps - 1;
        }
        field[tPosX][tPosY] = 'p';
        field[tPosX][tPosY] = 'x';
    }
    public static void showField() {
        int rivi = 0;
        while (rivi < field[0].length) {
            int sarake = 0;
            if (debug)
                System.out.println("sarake on" + sarake);
            while (sarake < field.length) {
                System.out.print(field[sarake][rivi]);
                sarake = sarake + 1;
            }
            rivi = rivi + 1;
            System.out.println();
        }
        System.out.println("Turtle is located at "
                + tPosX + "," + tPosY
                + ", moving " + tDir);
    }
}
```

D

Kuva 12 - Lähdekoodinäyte D. Uuden koodin lisäystä ja vanhan uudelleenjärjestelyä

Lähdekoodiin D (kuva 12) on lisätty uusia, täysin turhia kohtia. Esimerkiksi muuttujien kertomista yhdellä, else-if -haara johon ei missään tapauksessa päädytä, sekä kaksi tulostuslausetta, jotka suoritetaan vain mikäli muuttuja ”debug” on tosi, mitä se ei ikinä ole. Toisistaan riippumattomia koodisekvenssejä on järjestelty uudestaan: kaksi metodia on vaihdettu toisinpäin, else-if -haarojen paikkoja ja luokan attribuuttimäärittelyjä on sekoitettu.

```
public class E {
    private static char[][] field;
    private static char pen = '*';
    private static int tPosX = 1;
    private static int tPosY = 1;
    private static String tDir = "right";
    public static void showField() {
        for (int rivi = 0; rivi<field[0].length; rivi++)
        {
            for (int sarake=0; sarake<field.length; sarake++)
            {
                System.out.print(field[sarake][rivi]);
            }
            System.out.println();
        }
        System.out.println("Turtle is located at "
                + tPosX + "," + tPosY
                + ", moving " + tDir);
    }
    public static void moveTurtle(int steps) {
        for (;steps > 0;steps--) {
            field[tPosX][tPosY] = pen;
            switch (tDir) {
                case "up":
                    tPosY = tPosY - 1;
                    break;
                case "right":
                    tPosX = tPosX + 1;
                    break;
                case "down":
                    tPosY = tPosY + 1;
                    break;
                default:
                    tPosX = tPosX - 1;
                    break;
            }
        }
        field[tPosX][tPosY] = 'x';
    }
}
```

E

Kuva 13 – Lähdekoodinäyte E, silmukka- ja ehtorakenteiden vaihto toisiin.

Koodinäytteesä E (kuva 13) on while-silmukat korvattu for-silmukoilla, ja if-else -rakenteet switch-casella. Ohjelma F (kuva 14) sisältää kaikki näytteissä B-E tehdyt muutokset. Ohjelmat G ja H (kuva 15) eivät sisällä minkäänlaista mainittavaa yhtäläisyyttä muihin.

```
public class F {

    private static boolean debug = false;
    private static String suunta = "right";
    private static char[][] kentta;
    private static int Y = 1;
    private static char kyna = '*';
    private static int X = 1;

    public static void liikutaKonnaa(int askelia)
    {
        if (debug)
            System.out.println("@liikutus");
        for (;askelia > 0;askelia--) {
            kentta[X][Y] = kyna;
            switch (suunta) {
                case "down":
                    Y = Y + 1;
                    break;
                case "right":
                    X = X + 1;
                    break;
                case "ehehe":
                    X = X * 100;
                    break;
                case "up":
                    Y = Y - 1;
                    break;
                default:
                    X = X - 1;
                    break;
            }
        }
        kentta[X][Y] = 'x';
    }
    public static void naytaKentta()
    {
        if (debug)
            System.out.println("@tulostus");
        for (int i = 0; kentta[0].length>=i; i++)
        {
            for (int j=0; kentta.length>=j; j++)
                System.out.print(kentta[j][i]);
            System.out.println();
        }
        System.out.println("Turtle is located at "
                + X + "," + Y
                + ", moving " + suunta);
        if (debug) System.out.println("@lopussa");
    }
}
```

F

Kuva 14 - Lähdekoodinäyte E, kaikki aikaisemmat muutokset kumulatiivisesti

```
public class H {
    private int N;
    private Object[] A;
    private int n;
    public H() {
        N = 1;
        n = 0;
        A = new Object[N];
    }
    public boolean isEmpty() { return n == 0; }
    public int arraySize() { return N; }
    public void push(Object x) {
        A[n] = x;
        n++;
        if (n == N) {
            N *= 2;
            Object[] B = new Object[N];
            System.arraycopy(A, 0, B, 0, n);
            A = B;
        }

    }
    public Object pop() {
        if (this.isEmpty())
            return null;
        n--;
        if (N >= 2 && n == N / 4) {
            N /= 2;
            Object[] B = new Object[N];
            System.arraycopy(A, 0, B, 0, n);
            A = B;
        }
        return A[n];
    }
}
```

H

```
public class G {

   public static boolean onkoVokaali(char merkki) {
      if (merkki == 'a' || merkki == 'e' || merkki == 'i'
        || merkki == 'o' || merkki == 'u' || merkki == 'y'
        || merkki == 'ä' || merkki == 'ö')
         return true;
      else
         return false;
   }

   public static int laskeVokaalit(char[] t) {
      int voklkm = 0;
      if(t == null)
         return 0;
      for (int i = 0; i < t.length; i++)
         if (onkoVokaali(t[i]))
            voklkm++;
      return voklkm;
   }

   public static void kerroTulos(String nimi, int lkm) {
      System.out.println("Taulukossa \"" + nimi
               + "\" oli " + lkm + " vokaalia.");
   }

   public static void main(String[] args) {
      char[] taulu1 = { 'l', 'a', 'k', 'i' };
      char[] taulu2 = null;
      int lkm1 = laskeVokaalit(taulu1);
      kerroTulos("taulu1", lkm1);
      int lkm2 = laskeVokaalit(taulu2);
      kerroTulos("taulu2", lkm2);
   }
}
```

G

Kuva 15 - Lähdekoodit G ja H, ratkaisuja täysin eri tehtäviin.

4.2.2. Tulokset

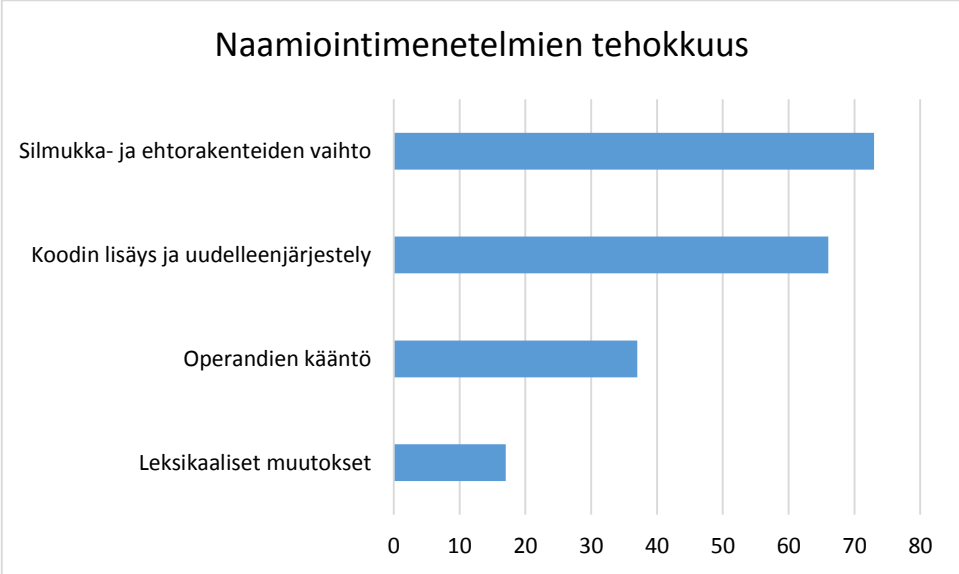
Tulokset on esitetty taulukossa 1. Niissä ei ole mitään kovin yllättävää. Limittäin olevia alijonoja etsivät SIM ja JPlag suoriutuivat selvästi parhaiten, JPlag vielä jonkin verran paremmin kuin SIM. MOSS ei hämääntynyt leksikaalisista muutoksista, mutta kaikki muut operaatiot laskivat sen antamaa samankaltaisuuslukemaa huomattavasti. Tekijöidensä mukaan MOSS välttää kaikin keinoin tuottamasta vääriä positiivisia, joten ehkä tässä on enemmänkin kyse erilaisista prioriteeteista kuin tunnistuksen ”huonoudesta”. Molemmilla Sherlock-nimisillä ohjelmilla oli vaikeuksia yhdistää plagiaatteja alkuperäiseen jo triviaalien muutosten jälkeen, koska ne pyrkivät täsmäämään myös tunnisteita ja kommentteja. Näytettä A ei osannut yhdistää näytteeseen F mikään muu kuin JPlag. Yksikään (oikeista) ohjelmista ei antanut vääriä positiivisia, eli ei merkannut näytteitä G tai H samankaltaisiksi minkään muun kanssa. TEST totesi kaikki tiedostot plagiaateiksi, myös kontrollinäytteet. Tämä johtuu epäilemättä siitä, että tokenisaatiovaiheessa käytetty kohdeaakkosto

on aivan liian suppea. Ongelman voisi yrittää korjata jakamalla avainsanoja ja ehkä operaattoreita useampaan kuin yhteen kategoriaan.

Kuvassa 16 on havainnollistettu eri naamiointimenetelmien tehokkuutta pylväsdiagrammilla. Tehokkuus on tässä mitattu sen mukaan, paljonko menetelmää edustavan lähdekoodinäytteen keskimääräinen samankaltaisuusprosentti erosi sadasta taulukossa 1. Silmukka- ja ehtorakenteiden vaihto osoittautui tässä tapauksessa hyvinkin tehokkaaksi, mutta se voi osaltaan selittyä sillä, että näytteen A koodista hyvin suuri osa koostui juuri näistä kahdesta. Suuremmalla otoksella koodin lisäys ja uudelleenjärjestely olisi mahdollisesti päätynyt kärkeen.

	A ↔ B leksikaalisia muutoksia koodiin	A ↔ C operandien vaihdoksia	A ↔ D koodin lisäystä ja järjestelyä	A ↔ E silmukka- ja ehto-rakenteiden vaihto	A ↔ F kaikki edelliset	A ↔ G eri koodi, pienempi tulos on parempi	A ↔ H eri koodi, pienempi tulos on parempi
MOSS	99	39	10	21	0	0	0
JPlag	100	100	41	24	14	0	0
SIM	100	67	30	35	0	0	0
Sherlock(1)	33	7	88	15	0	0	0
Sherlock(2)	85	100	0	40	0	0	0
TEST	100	97	86	96	86	76	86
(Keskiarvo%)	83	63	34	27	3	0	0

Taulukko 1 - Työkalujen laskemat samankaltaisuusprosentit aikaisemmin kuvatuille lähdekooditiedostoille



Kuva 16 - Naamiontimenetelmien tehokkuus mitattuna niiden aiheuttamalla keskimääräisellä samankaltaisuusprosentin putoamisella

Jälkikäteen tarkasteltuna näytteiden G ja H lisääminen testiin oli melko yhdentekevää – ne lisättin, koska alustavissa kokeiluissa näytti siltä, että jotkut ohjelmista löytävät yhtäläisyyksiä silmämääräisesti täysin eri tiedostoista. Kun vertailtavien tiedostojen pituutta kasvatettiin, tämä ongelma väheni huomattavasti. Voidaan todeta, että toimiakseen järkevästi, suurin osa työkaluista vaatii vertailtavien dokumenttien olevan yli kahdenkymmenen rivin mittaisia. Lisäksi huomattiin, että tarpeeksi lähdekoodia muokkaamalla saadaan kaikki kokeillut ohjelmat näyttämään lähellä nollaa olevaa samankaltaisuutta. Se ei välttämättä ole väärä tulos – näytteet A ja F ovat melko erilaisia, vaikka toinen onkin tuotettu toisen pohjalta. Jos nämä kaksi olisi nähty samaksi ohjelmaksi, voisi olettaa tunnistuksen tarkkuuden melko huonoksi, jos kaikki vertailtavat ohjelmat toteuttaisivat samaa tehtävänantoa.

Taulukkoa 1 ei tietenkään voi pitää kovin kattavana viitteenä järjestelmien hyvyydestä jo siksi, että vertailtujen ohjelmakoodien ei joukko vastaa kooltaan tai laadultaan tositilannetta. Tarkkuutta mittaavat näytteet olivat liian kaukana muista siinä mielessä, että ne eivät ratkaisseet identtistä ohjelmointitehtävää. Plagiarismi tässä tapauksessa oli myös täysin keinotekoista, eikä välttämättä vastaa oikeiden opiskelijoiden hämäystarkoituksissa tuottamaa. Monien kirjoittajien mielestä myös yksittäisten ohjelmien koko olisi varmasti edelleen liian pieni, jotta vertailun olisi mahdollista toimia täyttä potentiaaliaan vastaavasti. Toisaalta tässä käytetyt esimerkit (näytteet A, G ja H) ovat ratkaisuja oikeisiin Tampereen yliopiston perustason ohjelmointikurssin tehtäviin, joten siinä mielessä niiden koko on juuri sopiva. Samankaltaisia vertailuja ovat suorittaneet mm. Martins ja muut [2014] sekä Hage ja muut [2010].

5. Automaattisen tunnistuksen huijaaminen

Plagiaatintunnistimet – kuten edellä olemme huomanneet – eivät ole erehtymättömiä. Niitä on mahdollista huijata siinä missä ihmistarkastajaakin. Tässä luvussa esitetään tapoja, joilla niitä vaikuttaisi olevan kaikkein helpointa hämätä, ja annetaan niistä pari esimerkkiä (listalla kursiiveissa).

Gillam ja muut [2010] kuvaavat yleisiä luonnollista kieltä tutkivien plagiaatin-tunnistusjärjestelmien huijauskeinoja. Näitä on esimerkiksi merkkien korvaaminen toisilla, ihmissilmälle saman näköisillä merkeillä. Kirjain 'e' voidaan esimerkiksi korvata kyrillisellä 'e'-symbolilla. Koneellinen tunnistus on tällaiselle altis – eihän merkki ole enää ollenkaan sama – kun taas ihminen ei huomaa vaihdosta välttämättä lainkaan, eikä teksti näytä hänelle epäilyttävältä. Samaan kategoriaan voidaan laskea myös mahdollisuus korvata välilyönnit merkillä, jota ei esiinny muualla tekstissä, ja sitten värjätä em. merkki valkoiseksi (jolloin se ihmislukijalle näyttää taas tavalliselta välilyönniltä). Toinen keino on korvata sanoja synonyymisanakirjan avulla, tai kääntää teksti vieraalle kielelle ja mahdollisesti takaisin. On olemassa ohjelmistoja, jotka tekevät tämänkaltaisia naamiointitoimenpiteitä automaattisesti.

Onneksi meitä kiinnostaa ainoastaan lähdekoodin plagiointi, eikä mikään näistä metodeista ole kovin relevantti lähdekoodin tapauksessa. Seuraavien operaatioiden suorittaminen vaikuttaisi hämäävän kaikkia luvussa 4 kokeiltuja järjestelmiä. Lista on samankaltainen kuin Whalen [1990], mutta sisältää uusia kohtia ja jättää jotain pois. Lisäksi se on järjestetty naamiointimenetelmän sekä testatun että kirjallisuuden perusteella oletetun tehokkuuden mukaan laskevasti:

1. Vaihda kontrollirakenteita toisiin, vastaavan asian ajaviin (*if* ↔ *switch-case*, *for* ↔ *while*...)

2. Uudelleenjärjestele toisistaan riippumattomia koodisekvenssejä (kuten metodeja luokkatiedoston sisällä, if-else -haaroja, muuttujanesittelyjä, yms.).

3. Lisää ohjelmaan toiminnallisesti turhia kohtia. Ei välttämättä pitkiä, mutta paljon ja hajalleen, niin että alkuperäisen koodin kanssa yhteiset symbolijonot ja n-grammit vääristyvät mahdollisimman paljon.

4. Käytä tai poista käytöstä mahdollisimman paljon syntaktista sokeria: *i++* ↔ *i=i+1, x*=y* ↔ *x=x*y, if (x==y) z = ...* ↔ *z = (x==y ? ...*

5. Korvaa literaaleja tilapäisillä muuttujilla, tai toisinpäin: *i=3+4;* ↔ *k=4; h=5; i=k+h.*

6. Siirrä osia koodista funktioihin ja korvaa niiden alkuperäinen esiintymispaikka funktiokutsulla (vaikka sitä käytettäisiin vain kerran). Tai vastaavasti poista funktioita ja korvaa niitä kutsuvat kohdat niiden rungolla.

7. Uudelleenkäytä (tai poista uudelleenkäytöstä) muuttujia. Esimerkiksi silmukoiden laskurit tai muut tilapäisessä käytössä olevat muuttujat voi usein niiden käytön jälkeen kierrättää samassa skoopissa toiseen tarkoitukseen, sen sijaan, että esittelisi ja alustaisi uusia. Tämä vääristää jonkin verran sekä Halsteadin metriikoita että riippuvuusgraafeja.

8. Korvaa ehtolauseita niiden kanssa loogisesti ekvivalenteilla, mutta eri tavoin muotoilluilla lausekkeilla ja vaihda operandeja eri puolille operaattoria: *if (a || b) ↔ if (!(!b && !a))* jne. Esimerkiksi &&-operaattorin voi myös korvata kahdella sisäkkäisellä if-lauseella.

9. Poista tai lisää merkityksettömiä tyyliseikkoja, kuten aaltosulkeita yhden rivin if-lauseen ympärille. Vaihda kaikkien tunnisteiden nimet. Poista kommentit kokonaan (kirjoitusvirheiden toistamisen välttämiseksi) ja kirjoita uudet tilalle – eri kohtiin.

Myös tämä prosessi olisi varmasti mahdollista automatisoida. Voitaisiin siis toteuttaa ohjelma, joka ottaisi syötteenään lähdekoodin, tekisi sille nämä operaatiot (poislukien uusien kommenttien generoinnin) ja tuottaisi automaattisesti naamioidun plagiaatin alkuperäisestä lähdekooditiedostosta. Sen lisäksi, että tällainen voitaisiin nähdä jollain tapaa arveluttavana, se ei myöskään olisi millään muotoa triviaali toimenpide. Jotta edellä kuvatun ohjelman olisi mahdollista muokata lähdekoodia vaikuttamatta sen toimintaan, pelkkä approksimaatio koodin rakenteesta ei riitä. Toisin kuin esimerkiksi tokenisaatioprosessissa, missä pieni väärintulkinta ei dramaattisesti vaikuta tulokseen, kyseessä oleva ohjelmointikieli olisi välttämätöntä osata täysin jäsentää. Lisäksi, vaikka automaattista tunnistusta onnistuisikin automaattisesti hämäämään, tulos ei todennäköisesti näyttäisi siltä, että sitä tekisi mieli palauttaa omalla nimellään.

6. Yhteenveto

Lähdekoodin automaattista plagiaatintunnistusta on tutkittu paljon, mutta täysin ongelmatonta ratkaisua siihen ei ole löydetty. Ehkä tämä johtuu siitä, että ohjelmien samankaltaisuutta arvioitaessa on alun alkaenkin kyse hieman sumeasta logiikasta, jossa tietokoneita ei ole vielä saatu olemaan erityisen hyviä. Ominaisuuksien laskenta on ensimmäinen laajalti käytetty menetelmä, mutta se on altis monenlaisille huijausyrityksille, tuottaa vääriä positiivisia, eikä yleisesti ottaen tunnista osittaista plagiointia ollenkaan. Tokenisaatiota ja siitä syntyvien symbolijonojen vertailua hyödyntävät strategiat ratkaisevat monta ominaisuuksien laskennan ongelmaa, mutta toisaalta luovat myös uusia. Esimerkiksi laajamuotoinen koodin uudelleenjärjestely on niille vaikea asia, eikä hyvin pitkien tai hyvin useiden symbolisekvenssien vertailu ole nopeaa, mikä on johtanut ainoastaan n-grammeista muodostettujen ”sormenjälkien” tarkasteluun ja sitä kautta hieman pintapuolisempaan analyysiin. Lähdekoodin puu- ja graafiesitysten vertailuun perustuvat menetelmät tuottavat (kirjallisuuden perusteella) parempia tuloksia, mutta niihin keskeisesti liittyvä graafien isomorfismin tai puiden samankaltaisuuden tunnistaminen on laskennallisesti hyvin raskasta, tehden niistä ainakin toistaiseksi hankalia kovin suurten ohjelmajoukkojen vertailussa. Lisäksi niiden toteuttaminen ei ole suoraviivaista, ja tuen lisääminen uudelle ohjelmointikielelle aina työlästä. Plagioinnintunnistustyökalua etsivälle tätäkin konkreettisempi ongelma on, ettei yhtään niihin perustuvaa järjestelmää ilmeisesti ole olemassa.

Plagiarismin rajat eivät ole selkeitä, ja mitä tarkemmin vertailu suoritetaan, sitä enemmän saadaan pääsääntöisesti myös vääriä positiivisia, mikä puolestaan lisää tarkistukseen liittyvän manuaalisen työn määrää. Työkalujen ja menetelmien hyvyyttä ei voi arvioida vain niiden löytämien plagiontitapausten mukaan – yhtä tärkeää on väärien positiivisten minimointi. Plagiointiepäilyjen selvittely on kaikille osapuolille ikävää. Ideaalitilanteessa tietysti plagioinnista rankaisemiseen jouduttaisiin turvautumaan vain harvoin. Tarkempi ja ennenkaikkea yhtenäinen määritelmä ohjelmakoodiplagiarismille yhdistettynä tehokkaaseen automaattiseen seulontaan ja siitä tiedottamiseen saattaisi ennaltaehkäistä vilppitapauksia. Koneellisesti voi tutkia vain lähdekoodin samankaltaisuutta, mikä ei implikoi plagiarismia. Ohjelmallisesti kerätyt tulokset on aina tarkastettava käsin, ja tällöin tarvitaan selkeitä linjauksia siitä, minkälainen ohjelmakoodin kopiointi on sallittua.

Voidaan ajatella, että plagiaatintunnistuksessa on vain yksi ratkaisematon kysymys: ohjelmakoodille pitäisi löytää sellainen esitysmuoto, joka ensinnäkin pysyy mahdollisimman muuttumattomana huolimatta koodiin tehdyistä syntaktisista muutoksista, ja joka toiseksi on kohtuullisen helposti vertailtavissa. Mikään tässä tutkielmassa käsitellyistä tai lähdekirjallisuudesta löytyneistä menetelmistä ei tunnu täyttävän molempia kriteereitä. On hyväksyttävä, että tarpeeksi huolella plagioitua lähdekoodia ei ole mahdollista yhdistää alkuperäiseen, ja se on kaikilla havainnoitavissa olevilla tavoilla mitaten täysin uusi ohjelma.

Tyypilliset taitamattoman ohjelmoijan tekemät naamiointiyritykset voidaan kuitenkin tunnistaa melko luotettavasti. Suurin osa nykyisistä työkaluista käyttää jonkinlaista tokenisaatiopohjaista lähestymistapaa, ja se vaikuttaa toimivan hyvin. Leksikaaliset muutokset koodiin eivät hämänneet kuin kahta kokeilluista järjestelmistä, ja pienet rakenteelliset muutoksetkin tuottivat vielä suhteellisen suuren samankaltaisuuslukeman. Tämä on useiden aiheesta kirjoittaneiden mielestä akateemisiin tarkoituksiin aivan riittävä saavutus. Uudempi tutkimus on siirtynyt vaihtoehtoisiin lähdekoodin esitysmuotoihin, mutta tämä ei ole vielä suuressa mittakaavassa konkretisoitunut ohjelmistoiksi. Mikäli kuitenkin luvun 5 lopussa kuvatun kaltaiset ohjelmat jostain syystä yleistyisivät, plagiaatintunnistuksen olisi jotenkin vastattava uudenlaisiin haasteisiin.

Viiteluettelo

[Ahtiainen et al., 2006] A. Ahtiainen, S. Surakka, & M. Rahikainen, Plaggie: GNU-licensed source code plagiarism detection engine for Java exercises. *Proc. of the 6th Baltic Sea conference on computing education research: Koli Calling,* (2006), 141-142

[Ahola, 2014] M. Ahola, Vilpillä viisaaksi? H-M. Järvisen haastattelu, *Opettaja* **40** (2014)

[Aiken et al., 2003] A. Aiken, S. Schleimer & D. Wilkerson, Winnowing: local algorithms for document fingerprinting. *Proc. of the 2003 ACM SIGMOD international conference on Management of data,* (June 2003), 76-85

[Barrón-Cedeño & Rosso, 2009] A. Barrón-Cedeño & P. Rosso, On automatic plagiarism detection based on n-grams comparison. *Advances in Information Retrieval* (2009) 696-700. Springer Berlin Heidelberg.

[Berghel & Sallach, 1984] H. Berghel & D. Sallach, Measurements of program similarity in identical task environments. *ACM SIGPLAN Notices*, **19(8)**, (1984), 65-76.

[Culwin et al., 2001] F. Culwin, A. MacLeod & T. Lancaster, Source code plagiarism in UK HE computing schools, *Issues, Attitudes and Tools, South Bank University Technical Report SBU-CISM-01-02. (2001)*

[Cosma & Joy 2012] G. Cosma & M. Joy, An approach to source-code plagiarism detection and investigation using latent semantic analysis. *Computers, IEEE Transactions on,* **61(3),** (2012), 379-394.

[Demaine et al., 2009] E. Demaine, S. Mozes, B. Rossman & O. Weimann, An optimal decomposition algorithm for tree edit distance. *ACM Transactions on Algorithms*, ***6*(1)**, (2009)

[Dumais et al., 1988] S. Dumais, G. Furnas, T. Landauer, S. Deerwester & R. Harshman, Using latent semantic analysis to improve access to textual information, *Proc. of the SIGCHI conference on Human factors in computing systems* (May 1988), 281-285

[Faidhi & Robinson, 1987] Faidhi, J. A., & Robinson, S. K. An empirical approach for detecting program similarity and plagiarism within a university programming environment. *Computers & Education*, **11** (1987), 11-19.

[Ferrante et al., 1987] J. Ferrante, K. J. Ottenstein & J. D. Warren, The program dependence graph and its use in optimization, *ACM Transactions on Programming Languages and Systems,* **9(3)** (1987), 319–349

[Grune & Huntjens, 1989] D. Grune & M. Huntjens, Detecting copied submissions in computer science workshops. Informatica Faculteit Wiskunde & Informatica, Vrije Universiteit. (1989).

[Grier, 1981] S. Grier, A tool that detects plagiarism in Pascal programs. *ACM SIGCSE Bulletin*, **13(1),** (February 1981), 15-20.

[Gillam et al., 2010] L. Gillam, J. Marinuzzi, & P. Ioannou, TurnItOff–defeating plagiarism detection systems, *11 Th Annual Conference of the Subject Centre for Information and Computer Sciences,* **84**, (2010)

[Hage et al., 2010] Hage, J., Rademaker, P., & van Vugt, N., A comparison of plagiarism detection tools. Utrecht University. Utrecht, Netherlands (2010)

[Halstead, 1977] M. Halstead, Elements of software science. Elsevier Science Inc. (1977)

[Heckel, 1978] P. Heckel, A technique for isolating differences between files. *Communications of the ACM*, **21(4),** (1978), 264-268.

[Hirschberg, 1975] Daniel Hirschberg, A linear space algorithm for computing maximal common subsequences, *Communications of the ACM,* **18.6** (1975), 341-343.

[Joy & Luck, 1999] M. Joy & M. Luck, Plagiarism in programming assignments, *IEEE Transactions on Education*, **42(2)**, (1999), 129-133.

[Järvinen, 2014] M. Järvinen, Plagioinnin automaattinen etsintä lähdekoodista. Tampereen teknillinen yliopisto, Pro-Gradu -tutkielma (2014).

[Karp & Rabin, 1987] R. Karp & M. Rabin, Efficient randomized pattern-matching algorithms, *IBM Journal of Research and Development*, **31(2),** (1987), 249-260.

[Kondrak, 2005] G. Kondrak, N-Gram Similarity and Distance. *String Processing and Information Retrieval* (2005), 115.

[Levenshtein, 1966] V. I. Levenshtein, Binary codes capable of correcting deletions, insertions, and reversals. *Soviet physics doklady*, **10** (February 1966), 707-710.

[Ligaarden, 2007] O. Ligaarden, Detection of plagiarism in computer programming using abstract syntax trees, University of Oslo, Dept. of Informatics, Masters Thesis. November 2007.

[Liu et al., 2006] C. Liu, C. Chen, J. Han, & P. Yu, GPLAG: detection of software plagiarism by program dependence graph analysis. *Proc. of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, (2006), 872-881

[Löfström & Kupila, 2012] E. Löfström & P. Kupila, Plagioinnin syyt ja yliopisto-opetuksen keinot puuttua niihin. *Journal of University Pedagogy* (2012) Available: *http://lehti.yliopistopedagogiikka.fi/2012/05/03/plagioinnin-syyt-ja-yliopisto-opetuksen-keinot-puuttua-niihin/*

[Malpohl et al., 2002] G. Malpohl, L. Prechelt & M. Philippsen, Finding plagiarisms among a set of programs with JPlag. *J. UCS*, **8(11)**, (2002), 1016.

[Martins et al., 2014] V. T. Martins, D. Fonte, P. R. Henriques, & D. da Cruz, Plagiarism Detection: A Tool Survey and Comparison. *In: Proc. of 3rd Symposium on Languages, Applications and Technologies* (2014), 143–158

[Ottenstein, 1976] K. Ottenstein, An algorithmic approach to the detection and prevention of plagiarism. *ACM Sigcse Bulletin*, **8(4)**, (1976), 30-41.

[Ottenstein & Ottenstein, 1984] K. Ottenstein & L. Ottenstein, The program dependence graph in a software development environment. *ACM Sigplan Notices,* **19(5)**, 177-184.

[Parker & Hamblen, 1989] A. Parker & J. Hamblen, Computer algorithms for plagiarism detection, *IEEE Transactions on Education*, **32** (May 1989), 94–99

[Pawlik & Augsten, 2011] M. Pawlik, & N. Augsten, RTED: a robust algorithm for the tree edit distance. *Proceedings of the VLDB Endowment*, **5(4)**, (2011) 334-345.

[Sraka & Kaucic] D. Sraka, & B. Kaucic, Source code plagiarism. *Proc. of the ITI 2009 31st International Conference on Information Technology Interfaces, 2009*, 461-466

[Stapleton, 2012] P. Stapleton, Gauging the effectiveness of anti-plagiarism software: An empirical study of second language graduate writers. *Journal of English for Academic Purposes,* **11(2)** (2012), 125-133.

[Surakka et al., 2002] V. Surakka, M. Ihonen, J-M. Heimonen, P. Isokoski, M. Juhola, P. Majaranta, H. Rikala & T. Virtanen, Plagiointi opintosuorituksissa, *Plagiointityöryhmän alustava raportti, Tampereen Yliopisto,* (2002), Available: *http://www.uta.fi/sis/tkt/ohjeet/plagiointi/plagiointi.rtf*

[Tao et al., 2013] G. Tao, D. Guowei, Q. Hu, & C. Baojiang, Improved Plagiarism Detection Algorithm Based on Abstract Syntax Tree. *Fourth International Conference on Emerging Intelligent Data and Web Technologies* (September 2013) 714-719

[Tutkimuseettinen neuvottelukunta, 2012] Hyvä tieteellinen käytäntö ja sen loukkausepäilyjen käsitteleminen Suomessa, *ohje* Available: *http://www.tenk.fi/sites/tenk.fi/files/HTK_ohje_2012.pdf*

[Ukkonen, 1992] E. Ukkonen, Approximate string-matching with q-grams and maximal matches. *Theoretical computer science*, **1** (1992), 191-211.

[Verco & Wise, 1996] K. Verco, & M. Wise, Software for detecting suspected plagiarism: Comparing structure and attribute-counting systems. *ACSE*, (1996).

[Whale, 1990] G. Whale, Identification of program similarity in large populations, *The Computer Journal* **33(2)** (1990), 140-146.

[Wise, 1993] M. Wise, String similarity via greedy string tiling and running Karp-Rabin matching. *Online Preprint*, (Dececmber 1993).

[Wise, 1996] M. Wise, YAP3: Improved detection of similarities in computer program and other texts. *ACM SIGCSE Bulletin,* **28(1)**, (March 1996), 130-134.

[Youmans, 2011] R. Youmans, Does the adoption of plagiarism-detection software in higher education reduce plagiarism? *Studies in Higher Education*, **36(7)**, (2011), 749-761.

[Zeidman 2004] B. Zeidman, Tools and algorithms for finding plagiarism in source code, June 2004, Available at: *http://www.drdobbs.com /184405734* (Visited: 21.4.2015)

[Zhang & Shasha, 1989] K.Zhang & D.Shasha, Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal on Computing*, **18(6)**, (1989), 1245-1262.